AN ENHANCED BLUE-GREEN DEPLOYMENT FOR REDUCING COST AND APPLICATION DOWNTIME

H.M.D. THILINA JAYAWARDANA

168227V

MSc in Computer Science

Department of Computer Science and Engineering

University of Moratuwa
Sri Lanka

May 2018

DECLARATION

I declare that this is my own work and this MSc Research Report does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my thesis, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works.

…… ……

H.M.D.T. Jayawardana Date

I certify that the declaration above by the candidate is true to the best of my knowledge and that this project report is acceptable for evaluation for the MSc Research.

…… ……

Dr. Indika Perera Date

ACKNOWLEDGEMENT

I would like to sincerely convey my gratitude to my supervisor Dr. Indika Perera for the guidance and motivation throughout the research. Also my appreciation goes to University LIbrary for providing research materials. I am also thankful for my colleagues and the staff of Pearson Lanka Pvt LTD for their valuable help and support.

ABSTRACT

Application deployment is one of the critical milestones in the software development lifecycle. There are always risks of downtime and failing the new application version. Blue-Green deployment aka A/B deployment is one of the popular web application deployment techniques to mitigate those deployment risks. With the Blue-Green approach, it provides a quick backout plan with an existing set of servers with the previous application version up and running. Even though this has become more popular with the development of cloud infrastructure services, there are some scenarios still this approach brings disadvantages.

In this research, we discuss alternative development approaches in order to address above mentioned concerns while preserving the favorable features which are available in the Blue-Green deployment methodology. It has been considered two alternative approaches for the Blue-Green process without impacting the applications. It has been thoroughly analyzed each alternative approach that we suggest with in order to determine an alternative deployment process for the suitable situation.

Throughout this research, it has been considered Java web application deployment processes as the concerned scenario. As an alternative deployment processes, it has been discussed some of the already existing methodologies and trending novel techniques as well.

It has been proposed two alternative deployment mechanisms comparative to the Blue-Green deployment methodology. The first approach is proposed using the Parallel deployment capability of Apache Tomcat and the second approach is Deployment using Linux containers. Both of these approaches have been tested along with the conventional Blue-Green deployment methodology. The efficiency of each alternative approach has been assessed in a popular cloud environment Amazon Web Service (AWS) considering the practical usage of the solutions.

With this research it has been considered enhancing the existing Blue-Green deployment methodology with the proposed alternative approaches.

By analyzing the results it has been concluded that proposed alternative approaches can be used to enhance the Blue-Green deployment with some pros and cons.

Keywords: Cloud, High-availability, Deployment, Downtime, Release, Web application

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviation	Description
API	Application Programming Interface
AWS	Amazon Web Services
CD	Continuous Delivery
CI	Continuous Integration
DNS	Domain Name System
DR	Disaster Recovery
HTTP	Hypertext Transfer Protocol
QOS	Quality of Service
SLA	Service Level Agreement
TPS	Transactions Per Second

CHAPTER 1

INTRODUCTION

1.1 Application deployment

Software application deployment is a usual maintenance task in the software lifecycle. Deploying software applications without any impact to the customers is important for availability critical applications. Various processes and tools are being used to support the application deployment to make it smooth as possible without any outage or downtime. Blue-Green deployment is one of the popular software deployment processes which have number of advantages on reducing the downtime. In this research it has been considered alternative ways that we can enhance the Blue-Green or A/B deployment while reducing the cost.

It is to be noted that throughout the rest of the chapters, the terms Blue-Green or A/B deployment may use interchangeably.

1.2. Application downtime

Downtime of an application is a state where the application is not usable or unavailable and not functioning. This state leads to impact customer experience and ultimately loose revenue for the application and business. Therefore many application deployment processes are looking for high availability options. Number of reasons can cause to such a state of an application downtime. In this research, we are considering only the application downtime related to application deployment.

Application downtime can be divided into two categories as:

1. Planned downtime
2. Unplanned downtime

Planned downtime is mostly due to a scheduled maintenance [24] where all the precautions are taken to restore the service quickly as possible. Planned downtimes are common in case of in-house data centers. Planned downtime can be caused by a number of maintenance tasks [18]. For example:

- Network maintenance
- Hardware maintenance
- System/software upgrades
- Security patching

- DNS changes
- Electrical maintenance

The planned downtimes are scheduled as to minimize the client impact and the revenue loss. Clients can be notified beforehand the scheduled date and time. Even though these types of maintenance were expected in data center era, with the cloud architecture planned maintenance not necessarily cause a service outage. With the cloud infrastructure, it can architect in a such a way that it can handle planned maintenance tasks [16][17].

Whereas unplanned downtime makes a software system unusable due to a number of reasons including power failures, hardware failure, application failures or defects and also human errors [1]. Since unplanned downtimes are unpredictable in nature, the impact tends to be higher than the scheduled maintenance downtimes. These events come to the customer with no precaution or notifications and therefore they directly impact the customer experience and revenue.

It can be seen that the cost of application downtime has been increased within past few years. Even a small unplanned outage of a critical application may affect many areas as cost i.e. third parties, end-users, lost revenue, business disruption, equipment, recovery, detection etc. [8].

Figure 1.1 depicts that cost of the outage has been increased over the years.

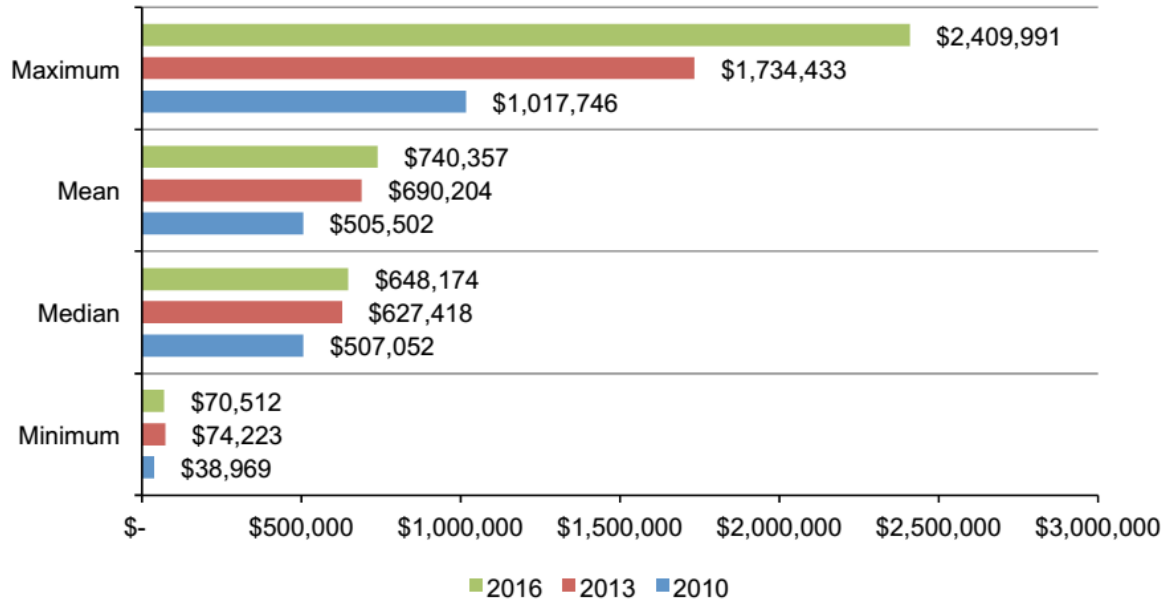

Figure 1.1 Cost of unplanned outage in data centers [8]

It should always be ready for outages of application services with quick rollback methods to reduce the impact of downtime. Longer the downtime more the cost. Following graph depicts the cost of unplanned downtime of minutes and the related cost.

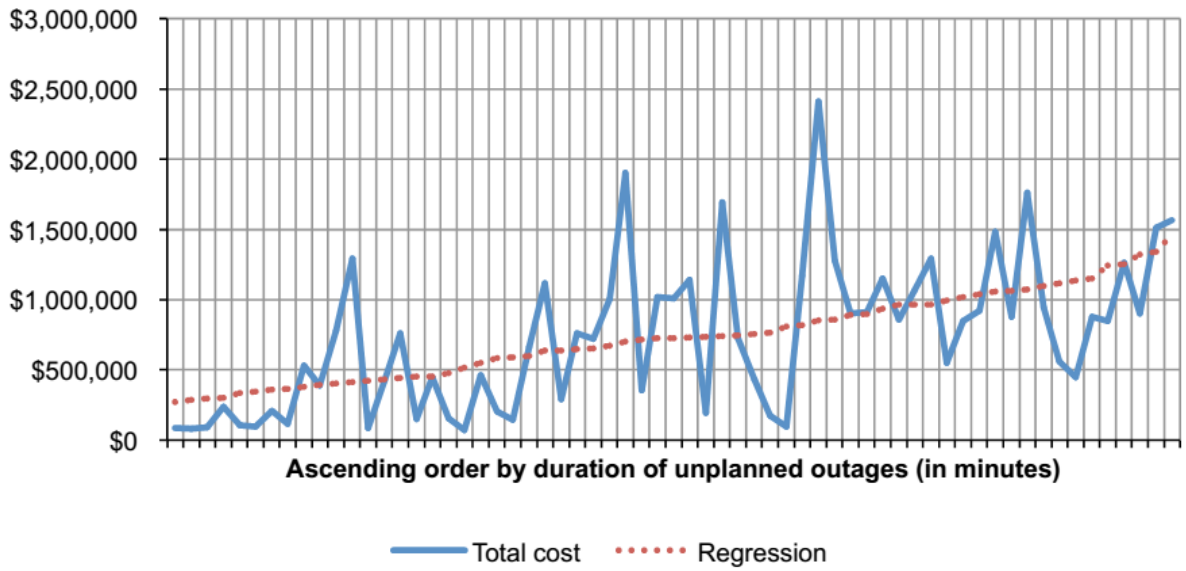

Figure 1.2 Cost vs duration of unplanned downtime [8]

1.3. Deployment downtime

Application deployment is one of the scheduled maintenance tasks which inherently comes with the risk of failures or a probability of a downtime. There is a number of deployment mechanisms available to mitigate those risks of application deployment and this is an area which continues evolving [26]. In this research, we are considering only on the downtime related to application deployment.

In terms of web applications, deployment is one of the important and critical milestones. In the deployment process of web applications, it should be always planned for risks of failing. When it is a critical application software, the planned downtime of an application deployment should be small as close to zero. Virtually this is considered as unnoticeable to the end-user of the system.

There is a number of mechanisms to quickly deliver software services to production using Continuous Integration (CI)/ Continuous Delivery (CD) approaches. A/B or Blue/Green deployment is one of the most popular techniques to deliver applications. This has become more popular with the development of cloud infrastructure services, many cloud providers encourage A/B deployment process. With A/B deployment, we can minimize the application downtime as well as it provides a way of quick rollback to previous version.

With the cheaper and easy to use cloud infrastructure, A/B deployment has been a promising way of deployment with minimum downtime yet there are still some drawbacks to this technique.

1.4. The problem and motivation

A/B or Blue-Green deployment brings a number of advantages to the deployment process to leverage the risks.

A/B deployment requires two sets of identical infrastructure maintained until decommissioning of the old set of servers. In the typical deployment scenario, old nodes are kept without decommissioning until it is confirmed the new set of nodes and the application version are stable. After decommissioning the older nodes, the rollback would be another release of course.

With this approach, keeping another version parallel to the live applications will be expensive since it is needed to have an exact copy of the infrastructure. Therefore this approach is not suitable when it requires keeping more than one application versions standby.

Switching to the new version is done by rerouting the live traffic pointing to the new application nodes. This is typically done either at load balancer level or DNS level. In session oriented scenario where load balancer uses sticky sessions when routing the traffic, it will always send the requests with the same session to the same node. In this situation, switching the traffic to new nodes might lead to temporary downtime to the clients. For a legacy application which does not use a common session store, this will be a problematic scenario.

At the time of traffic cutover, already connected clients may experience a moment of downtime

since the traffic is brutally disconnected from the older set of servers without properly completing the client request. `

In this research, we have addressed above-mentioned drawbacks of A/B deployment process by proposing a solution that is capable of solving these problems while preserving the positive properties of A/B deployment process.

A/B deployment process has been quickly adapted Continuous Integration/Continuous Deployment (CI/CD) mechanisms where it can be automated the deployment process. Typically it is automated the server creation, package deployment, testing up to traffic cutover.

In this type of deployment process, spinning up application nodes is a time consuming task. If it consists of several application nodes, this whole process can take several hours to complete.

Considering a fully automated CI/CD If something goes wrong the deployment environment it makes take much more time to investigate the problem and redeploy everything [19].

Even though Blue-Green deployment is promising so many advantages, still there are legacy applications which are stuck on the same server and unable to move into continuous deployment processes due to various reasons. Sometimes reusing the same server instance is mandatory considering the other project requirements and budget. Some of the legacy applications are unable to convert into A/B deployment model. For that kind of projects, it is expected a different deployment process which is cheaper and supports quick rollback and smooth cutover mechanism [19].

Above mentioned drawbacks of A/B deployment process motivated us into proposing a solution as a web server where it has the positive capabilities of A/B deployment in terms minimum downtime and quick rollback capabilities.

Above mentioned drawbacks of Blue-Green deployment process have been motivated to find a new deployment mechanism which can avoid those drawbacks while securing the smooth deployment and quick rollback capabilities.

1.5. Outline

The arrangement of the research is as mentioned below. It will be discussed in more details in each chapter separately.

Chapter 1 Introduction: A basic introduction of the research is mentioned in this chapter along with the problem and the motivation of the research.

Chapter 2 Literature Review: This chapter includes a detailed discussion of prior work related to the area of the study.

Chapter 3 Methodology: High level architecture and the evaluation plan and the experimentation steps are mentioned in this chapter.

Chapter 4 Solution Architecture and Implementation: This chapter is discussed about the detailed architecture and implementation of the solution.

Chapter 5 System Evaluation: Detailed evaluation of the solution and the results are discussed in this chapter.

Chapter 6 Conclusion: Study limitations and the conclusion has been discussed in this chapter based on the results observed.

1.6. Virtualization

Clustering and virtualization technologies are heavily used to support high availability of applications services and reduce the service downtime. Many researchers have focused on virtualization technologies such as Linux containers [5].

When considering the system downtime SLA is an important factor. When talking about planned downtime, it is important to make the service up and running within the SLA. SLA aware deployment [3] is a concept that has been studied with some of the researchers.

1.7. High availability architecture

Unlike the era of self-maintained data centers, with the facilities provided by cloud services the organizations do not require to bear the burden of maintaining the physical infrastructure.

Using local data centers can lead to a single point of failure when considering the server locations and the network. Addition to that, this is highly critical in case of a disaster situation. There is a high risk of losing the all the business critical functionality and the information in case of a disaster. Therefore, cloud service providers take a major part in disaster recovery requirements.

Cloud service providers give the capability of provisioning servers in multiple geographical locations and multiple availability zones. With this, there is no more single point of failures in the data center. Also, this architecture is inherently prone to disasters and providers better disaster recovery (DR) capabilities [16], [18], [19].

Some researchers are taking a step ahead of depending on a single cloud provider and studies on running the infrastructure with multiple cloud service providers [17]. This is more advantageous when considering the business critical and financial critical applications.

There can be some specific cloud features that only a single provider is capable of providing at the moment with their specialty of the area. Businesses who would deploy the applications against multiple cloud service providers would have the ability to use above mentioned specific areas from all the cloud providers.

1.6. Objectives

The objective of this research is to introduce a new deployment release process with following attributes.

Reduce deployment downtime

Downtime of the system should be minimum as possible. Therefore this is one of the critical objectives of this research.

Quick rollback and versioning

Deployment process should support a quick rollback mechanism. In case failure of the new application version, it should be possible to quickly rollback to the previous version with minimizing the system downtime. In order to identify the rollback version, the process should support a way to keep track of the application versions.

CI/CD support

New application deployment process should support CI/CD (Continuous Integration / Continuous Delivery) integrations. With the practice of agile processes and cloud oriented architecture, CI/CD plays a major role in software deployment. Therefore the proposed methodology should support with CI/CD tools.

CHAPTER 2

LITERATURE REVIEW

2.1. Prior work

Existing mechanisms to minimize the application downtime mostly talks about the unplanned downtime. An enterprise software is not a single software application, but they are consist of many applications including database, proxies, queues etc. There are many efforts related to each item of a system.

Reducing application downtime is not a new topic and there has been progressing in this area with many technologies. Engineers and researchers have been trying to reduce the application downtime in many aspects. A number of researchers have been conducted covering many aspects to reduce application downtime. There have been studies on both planned and unplanned system downtime.

2.1.1. SLA-Aware application deployment

System downtime is always bound with minimum value agreed upon SLA (Service Level Agreement) [3]. Therefore application deployment process should also adhere to the SLA. A heuristic approach to schedule application deployment has been proposed by SLA-Aware Application Deployment which they have considered SLA related matters for deployment process [3].

Failing to meet the agreed SLA is causing loss of profits in various direct and indirect ways including penalties.

Even though there are various studies on keeping single SLA parameters to provide quality of service (QoS), keeping the quality of service with multiple SLA parameters is hard. In the reality multiple SLA parameters (such as CPU, bandwidth, response time etc). should support the application services. With this research, they have studied on supporting multiple SLA parameters with scheduling heuristics. With this, they have implemented a new heuristics based load balancing mechanism.

2.1.2. A/B Deployment process

Basic idea behind A/B (aka Blue-Green) deployment process is switching live traffic between two identical environments running different application versions.

Typical A/B model would consist of two set of servers nodes.

A/Green - Live set

B/Blue - Stand by set

Though there are situations the live set and standby set were referred in different labels. The basic idea is creating a new set of server nodes with the new application version and switching the traffic to pointing to the new set. The old set of servers then become the “B” set whereas the current live set is referred as “A”.

Figure 2.1 depicts a typical A/B deployment where it switches the live traffic to the new environment from the load balancer.

Traffic cutover can be even done using an internal load balancer such as a proxy server. Most of the cloud infrastructure providers provide facilities to use a proxy or regional load balancers such as ha-proxy [1].

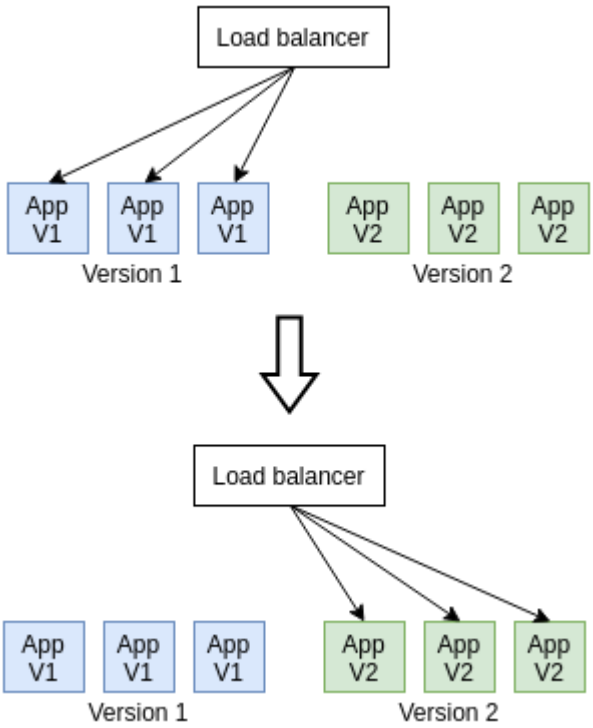

Figure 2.1 A/B deployment

When switching the traffic from the load balancer, one new application is added to the load balancer while removing an old node. It is normally done one node at a time to minimize the risk of failure. If new application nodes do not run well with the live traffic, if one node fails when it is first added to the load balancer, still the system can work with remaining old nodes.

Also, it is possible to shift the traffic from DNS level as well. In such a scenario, it is not needed to reconfigure the load balancer but the configuration would be in the DNS record. DNS record will be reconfigured pointing to the new environment [1].

Using DNS to switch the traffic may bind with the risk in case of failing the new application version. Unlike switching from the load balancer, DNS turns live traffic to new cluster at once [1], [2].

2.1.3. Canary deployment process

Canary Deployment is a deployment process similar to Blue-Green deployment where it forwards a small amount of live traffic to the new environment [1], [2]. This technique is used to test the new application version against the live traffic and this testing is called as Canary analysis or Canary testing [8]. Cloud infrastructure providers such as AWS also support to Canary testing approach with the deployment.

Here, only a small amount of live traffic is sent to new application version where it can be tested with live users. If the new version is working fine, then whole traffic can be routed to the new version.

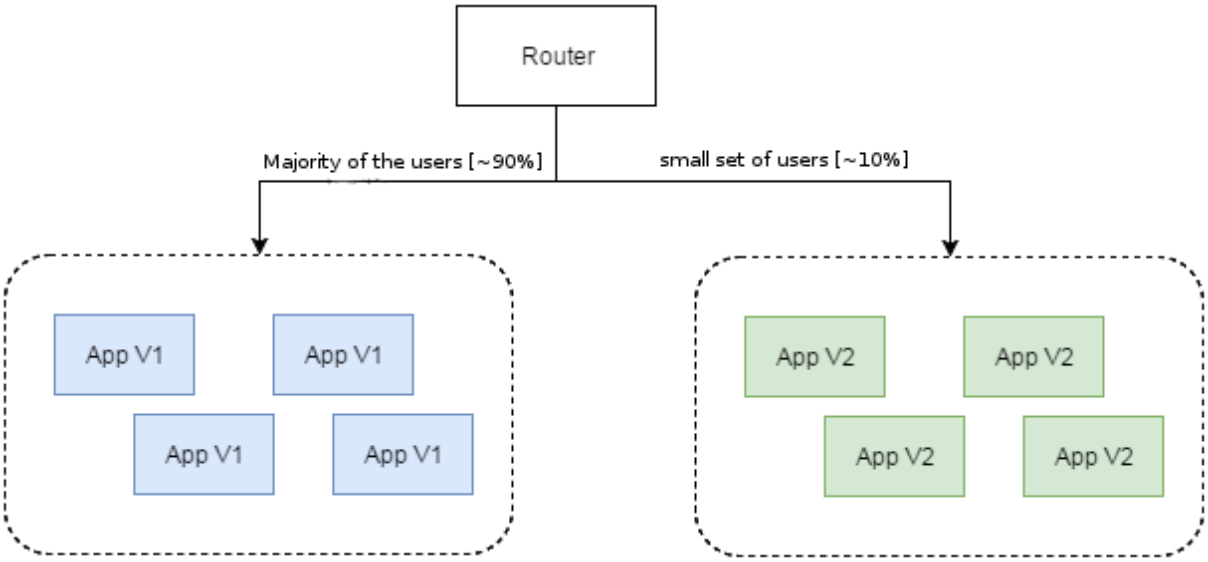

Figure 2.2 Canary deployment

With the Canary release, there is a chance to test the new version against the actual users without risking the whole user base of the system. A small part of the user base can be used to verify the stability of the new version.

If an issue identified during the canary testing, still it is possible to reroute the entire traffic to old stable version. In this aspect, this is very similar to the blue-green deployment process where it supports quick rollback capabilities. The idea of canary releasing is not to risk the entire user base.

2.1.4. Continuous delivery

Continuous delivery is a way of software development by enabling releases through build, test and deployment automation [25]. Continuous delivery makes a software product evolve by keep it updated with new releases. Hence development process should be consist of many shorter software development cycles. This concept highly supports agile processes. Agile processes help to constantly accept changes thus it requires frequent releases of the software product. Therefore continuous delivery has become popular with the agile software development process since it suits well with the process. Continuous delivery assures that software product can be released any time in a reliable way. This mechanism is mostly based on deployment pipelines [8][10].

Continuous Delivery is also bound with DevOps concept. DevOps is an idea of smooth interaction between development and operations of a software product.

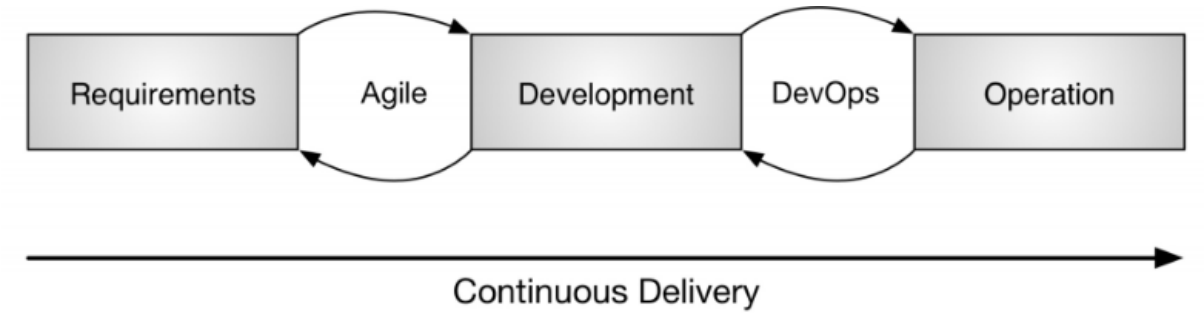

Figure 2.3 Continuous Delivery Process

Since most of the organizations favor cloud based infrastructures, a number of tools have been introduced for various purposes of the continuous delivery process.

Following are few examples of the tools [10]

Puppet - Configuration management
Vagrant - Virtualization of development environment
Jclouds - Java abstraction for cloud API
Whirr - Deployment Orchestration Engine
Brooklyn - Multi cloud deployment and management framework
Docker - Platform for portable distributed applications
Mesos - Cluster management

Apart from these, Jenkins, TeamCity, Travis CI are some CI tools mainly used in the industry.

These tools are used to automate the recursive parts of the software delivery process. Unit testing, packaging, cluster creation, application deployment, integration testing are few tasks that can be automated using above tools.

2.1.5. High-available cloud infrastructure

Cloud infrastructure services made it simple to virtualize a whole environment. It has been conducted many researchers in the area of increasing the availability of applications with virtualization technologies [5]. In the cloud environment, the application nodes are created as a cluster for the sake of high availability. Deploying a cluster with an automated script will be a complex and time consuming task. There are a number of technologies available for server node provisioning and clustering such as puppet, chef [2], [4]. With regards to CI/CD paradigm, our proposed solution will also be flexible to align with deployment automation complying with existing tools and technologies.

Container services are getting popular as a breakthrough of virtualization technologies. There are many researchers conducted with the user of container services to increase the availability of cloud applications [2], [1].

2.1.6. Multi-cloud scenarios

With the maturity of cloud provider giants and their saturated services, there is a trend among businesses to use multiple cloud provider instead of limiting to a single cloud. This way they can experience and take advantage of specialized features that can be only delivered by specific cloud providers [16].

Among this kind of researchers, one interesting approach is provisioning the infrastructure within multiple cloud platforms and get the benefit of cost reduction from both services [21]. Also, data transmission among cloud services is taking a bigger concern in this area as well.

With the use of multiple cloud platforms, a system can be prone against cloud provider specific organized cyber-attacks. Within past few years, we have been experienced few incidents as such.

2.1.7. Database downtime

For different components, there has been conducted researches to mitigate the unplanned and planned downtime. Out of that, a database is a crucial part of a system and a considerable amount of research has been conducted in that area to reduce the downtime of database applications. Also there could be found many patents in this area specifically for databases [6], [7].

Since the availability is a critical factor for the databases, most of the industrial database management systems support clustering and high availability with their architecture.
For example: Oracle, Cassandra, MySQL and MongoDB

Instead of using the self-hosted database clusters, there is a trend on a database as a service (Daas) type cloud services. Where the provider is taking care of the physical database server maintenance including hardware, security, performance and the availability [20]. Changing to cloud database services is favorable for the businesses on the aspect of maintenance overhead. With these services, they can concentrate on actual business functions instead of worrying about the database maintenance and the availability. For example: Amazon Dynamo, Amazon RDS and MongoDB Atlas

At the time of application deployment, database and data migration is a critical step. Since it is somewhat difficult to completely do a data migration for new application deployment, most cases the database is reused with the new application version. In such a case where it requires a data migration, there is a higher chance for service downtime due to the unavailable state of the database. It has been conducted much research on minimizing the downtime due to database migration [6], [7].

In this research, we are not considering the database deployment or database migration step of the application deployment process and we only consider the minimizing the downtime of application deployment in our proposed solution throughout the rest of this paper.

2.2. Literature review summary

As discussed above, this is a wide area of study with various research and practices focused on application availability and maintenance. As it has been studied the prior work related to this area, it can be seen that research projects focusing on the specific area of application deployment is quite narrow.

Considering the existing work related to this, it can be noticed that improving the application deployment mechanisms would be quite useful in the industry. As of now the current industry is investing more and more on high available applications and therefore, this research can be considered as a more of a support for small and medium sized companies and startups because of the improvements for the low cost application deployment methodologies.

CHAPTER 3

METHODOLOGY

3.1. Architecture and scope

A Java based simple web application was considered as the system under test in order to develop the solution. Tomcat server was chosen for this implementation since it is one of the widely used servlet container for simple web applications.

Typical A/B deployment process for the above application will consist of following steps.

1. Building and packaging
2. Executing unit tests
3. Provisioning server nodes
4. Package deployment
5. Executing automated integration and regression testing
6. Switching traffic to new nodes or the cutover

In an agile and Devops supporting environment, all of the above steps are expected to run automatically using CI/CD tools. The solution is to be implemented as it is supporting to agile and DevOps practices.

3.1.1. Versioning

In the A/B deployment process, one version of the application should be deployed in a similar environment as the production live server nodes. Therefore it is fair to say versioning is expensive in the A/B deployment process.

Rather than keeping more server nodes to keep the multiple versions, we propose to modify the process to be included with the capability to keep multiple application versions standby.

Tomcat 7 has introduced the notion of versioning of web applications by introduced parallel deployment feature [11]. This feature is taken as an advantage and is to be modified to make it more aware of the application version.

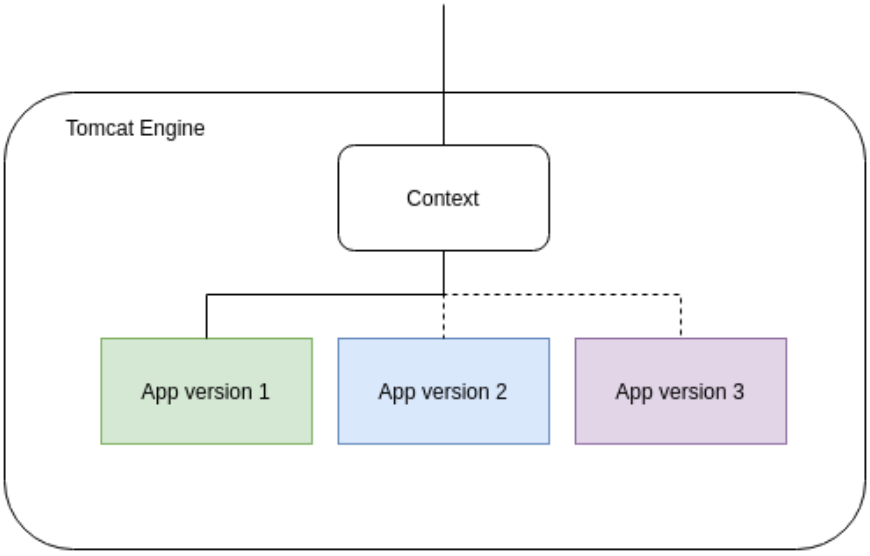

Figure 3.1 Multiple application versions running in Tomcat

Multiple versions of the application can be kept on the web server itself with some modifications. External API will be provided to switch the active context of the application. With this, it can easily change the live application version.

At the cutover point of the new version, there is a time where both versions of the applications running at the same time. This is handled by Tomcat making both application contexts live at the same time.

When running couple of versions of the same applications, application configurations may need to consider. If there is a configuration change between the two versions, the right configuration will be handled by the deployment process. Configurations of each version should be isolated from each other.

Logging is one of the major consideration when running the same application with many versions. If the same log path is given to both application versions, there may be some trouble accessing log files to write the records. In order to avoid this conflicts, Deployment process will manage the log file path of the application versions.

3.1.2. CI/CD Integration

Above mentioned modifications will also support CI/CD approaches in the release process. With the modifications, new API is exposed to deployment, rolling back, switching traffic, activating a new version etc.

These APIs are consumed by a client app integrated with the CI/CD tools. Jenkins platform is chosen as the CI platform and a plugin is developed to consume the API and integrate with the deployment process.

Jenkins post build plugins were used to invoke the above API endpoints. A new application has been developed to implement the necessary API endpoints. These endpoints are currently accessible by anyone and have not considered security aspects at this moment. Therefore it is assumed these endpoints are accessed only from the internal network and secured by the internal firewall.

Securing the deployment process is assumed to be out of scope for the moment.

3.1.3. Deployment process

With this deployment process, it is possible to use the existing application nodes without creating new ones for each and every application version. Provisioning of application nodes is not needed in the new process.

Unlike A/B release model, keeping multiple application versions is not as expensive as replicating the whole environment.

Deployment time is much reduced because it is not working with physical nodes and operating systems.

A smooth transition can be made with the cutover of the new application version. When activating the new application version, active user sessions can be shared by multiple application versions since all are the same engine. Web server is to be modified to support this

feature. Therefore it is not brutally cutting off active sessions of the user while deployment process. Therefore users will not experience any downtime of the system.

Rolling back to an earlier application version is also can be done via new APIs. If an issue found with the new application version, it can be deactivated and old version can be activated. Again in the rollback process also it will not make any impact to the user by providing a smooth transition.

The canary release process is also supported by the new modifications. Before doing the cutover, it will be able to define the routing percentage with the APIs.

3.1.4. System under test

One of the main concerns of developing this application is to keep the process independent of the application under test. Therefore the application under test was developed as simple as possible. This application is a JSP servlet application built using Spring MVC framework.

Spring MVC is a widely used Java application framework in the industry. Therefore this can be considered as a more suitable sample application for this research. Maven was used for the dependency management and packaging.

Since Maven tool is supported by Jenkins continuous integration pipelines, this can be automated easily. Compiling, packaging, versioning, package uploading to the repository also automated with the use of Jenkins maven plugin.

A war file is produced as the packaged artifact from the maven. This artifact is then pushed to the repository. In this case, AWS S3 bucket was used instead of an external maven repository considering the cloud environment. Since AWS is chosen for the cloud environment to test out the deployment process, using AWS S3 is obviously the best place to keep the file repository [1].

The WAR (Web Application Resource or Web application ARchive) file is deployed using a Tomcat servlet container.

The application has mainly three user interfaces:

Login

Home

Logout

The application does not contain complex UI interfaces since the frontend is not considered in this kind of deployment perspective. Therefore minimal required interfaces were used for this. General functionality used for each user interface.
Login: provider username and password and sign into the application
Home: This will go to a home interface and run a for loop in order to consume some processing power.
Logout: This logs out the user from the session.

The application was simply developed in order to act like a real work behavior and therefore sessions were used in order to keep the login state information.

3.2. Evaluation

Evaluation can mainly focus on deployment time and the system downtime. A simple application is chosen and it can be used to compare the two deployment models. The same application is deployed with the Blue-Green deployment method and also with the new solutions.

Time which was taken to complete the deployment process is measured and recorded.

Multiple requests are sent to the application continuously at the deployment time. The JMeter application is used to generate and control the load sent to the application.

With each deployment processes, Jmeter is executed multiple requests in a parallel manner. Each request is recorded with the completion time duration and the HTTP status code. A number of error codes are measured and compared against two deployment processes.

Results of above evaluations are compared and analyzed to materialize the conclusion of the research.

CHAPTER 4

SOLUTION ARCHITECTURE AND IMPLEMENTATION

4.1. Architecture Introduction

In this chapter, it has been discussed primarily the detailed architecture of the proposed solutions.

The overall solution has been discussed as two distinct approaches. Each approach is implemented in a similar environment. The environment is orchestrated similar to a general purpose production setup of a typical web application. Also considered multiple availability zones as well.

According to the selected evaluation methodology, client impact has also been captured with a use of simulated load for the application.

In order to test the above mentioned two approaches, a sample application has been developed in order to demonstrate a real scenario.

Each proposed architecture is considering the enhancements that can be done to the Blue-Green deployment methodology in the aspect of cost reduction.

4.2. Language and platform

Java technical stack is a high demand and highly used stable platform in the industry. When considering the enterprise services. JSP servlet technology has taken a huge part in the industry. Most of the novel web-based Java solutions are based on the servlets. Because of the robust and lightweight nature of servlet technology, a number of frameworks have been developed in order to support this.

In order to test this, we have developed a demonstration application. This application has been developed to run on Java version 1.8. This platform has been selected considering the stability and novelty of this Java platform version.

Similarly, this technology is supported by a community is driven and open source environment and has grown up to be a stable platform.

A number of servers also support JSP servlet technology which is commonly called as servlet containers. Out of all the supporting servlet containers, Tomcat is one of the most popular

server application. We have selected Tomcat 8 [11] as the target web server to deploy an application with the new deployment method.

4.3. Deployment method

Our attempt is to implement a deployment strategy where it can preserve following properties of A/B deployment methodology.

Reduce deployment downtime
Keep multiple versions ready
Easy rollback

Unlike in A/B deployment, our idea is to reduce the number of servers required for the deployment. Without spinning up new servers every time in a deployment, as the conventional deployment, we preferred to keep the physical server static.

We have designed two types of alternative deployment methods using tomcat servlet container as the target web server.

Mainly we have considered two approaches to be evaluated against A/B deployment.

A/B deployment

We have evaluated the A/B deployment along with our evaluation criteria since we are trying to propose a different and alternative approach for the popular A/B approach. Same criteria will be used to evaluate the alternative approaches as well.

Later the evaluation results would be used to synthesize a proper conclusion from this research.

Parallel deployment

As the first approach, we have used a single Tomcat server. In this case, we have taken the advantage of parallel deployment feature [11] that is available in the Tomcat server itself. Basic idea is to leverage the deployment and package versioning to the web server itself rather than externally manipulating the environment.

Using Linux containers

As the second method, we have used Linux container-based approach to deploying multiple tomcat instances in the single physical server.

Even though there are multiple Tomcat servers used it has been designed to minimize the impact over CPU and memory load with the help of Linux containers.

Addition to that, Linux containers have been used in order to resolve some of the disadvantages which may occur in the other deployment methodologies.

4.4. Approach 1 - Parallel deployment with Tomcat server

In this approach, we have used only a single Tomcat server. Tomcat itself has this parallel deployment feature that enables to keep multiple instances of the same application as in different versions. In this case, we have taken the advantage of parallel deployment feature [11] and designed a proxy component ahead of the tomcat server to interface the deployment process.

4.4.1. Parallel deployment

Parallel deployment is a feature that was introduced in the Tomcat version 7.0. With this feature, it is capable to manage multiple application versions without impact to the customers. This feature allows multiple contexts to be running in the same servlet container with different versions. Still, they will share a common pool of application sessions. Therefore the clients will not face any difficulties with their user experience.

Addition to that the proxy interface is capable of deploying, undeploying and rolling back the applications as required.

Since this approach covers the main objectives of this research project, we considered this to be evaluated against the conventional A/B deployment approach.

4.4.2. Proxy interface

As depicted in Figure 4.1, we have used a proxy application to interface Tomcat server for the deployment.

We have used a Nginx [15] server as the proxy interface to the Tomcat server. A simple web application is running in the proxy to communicate with Tomcat server.

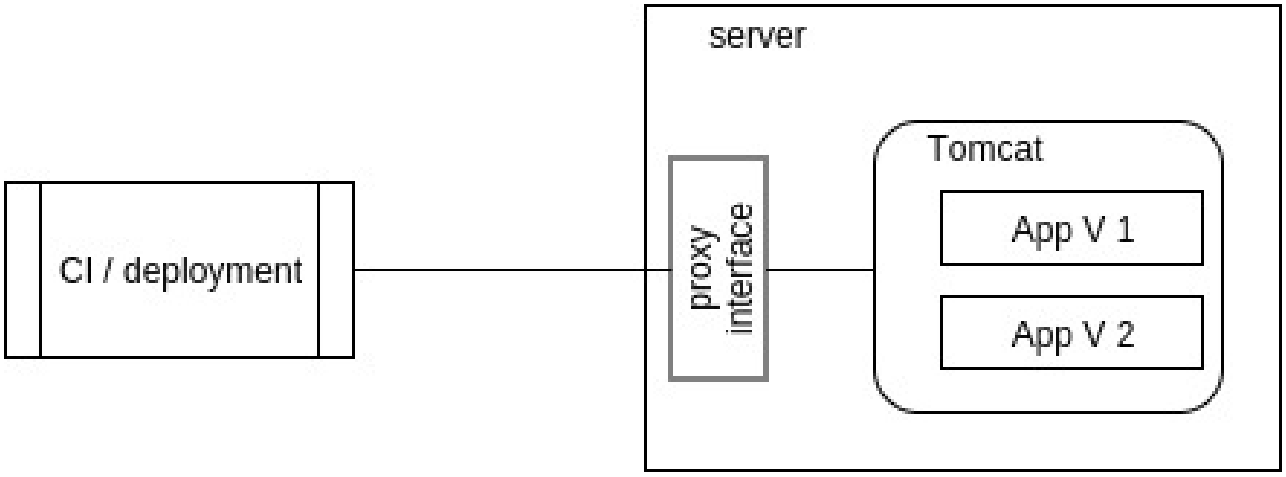

Figure 4.1 Approach 1 - Single Tomcat server

Tomcat 8 comes with a management application which provides facilities to deploy/un-deploy/start/stop applications. Proxy is communicating with Tomcat's manager app to deploy and un-deploy web applications.

Apart from that proxy interface do the versioning of the applications before sending them to deploy in Tomcat server. In this way, Tomcat can keep track of multiple versions of the application in the same Tomcat instance.

It is important to mention that, the above-mentioned proxy interface is only for deployment purposes but not to handle actual traffic for the tomcat server. Addition to that it also provides a graphical user interface which features deploy/undeploy/start/stop applications.

Also, it can use to keep track of currently active sessions for a particular application. JMX channels are used to communicate with Tomcat in order to get the active session information.

Apart from the GUI, it also provides an API for the same functions that can be used in CI/CD jobs in a Jenkins integration.

At the moment we have not considered about securing communication of Tomcat and the proxy or the Jenkins assuming all these components are not publicly accessible as in behind the firewall. Securing the internal communication will be considered as future improvements.

4.4.3. Clustering

This design has been setup in the AWS infrastructure as a cluster.

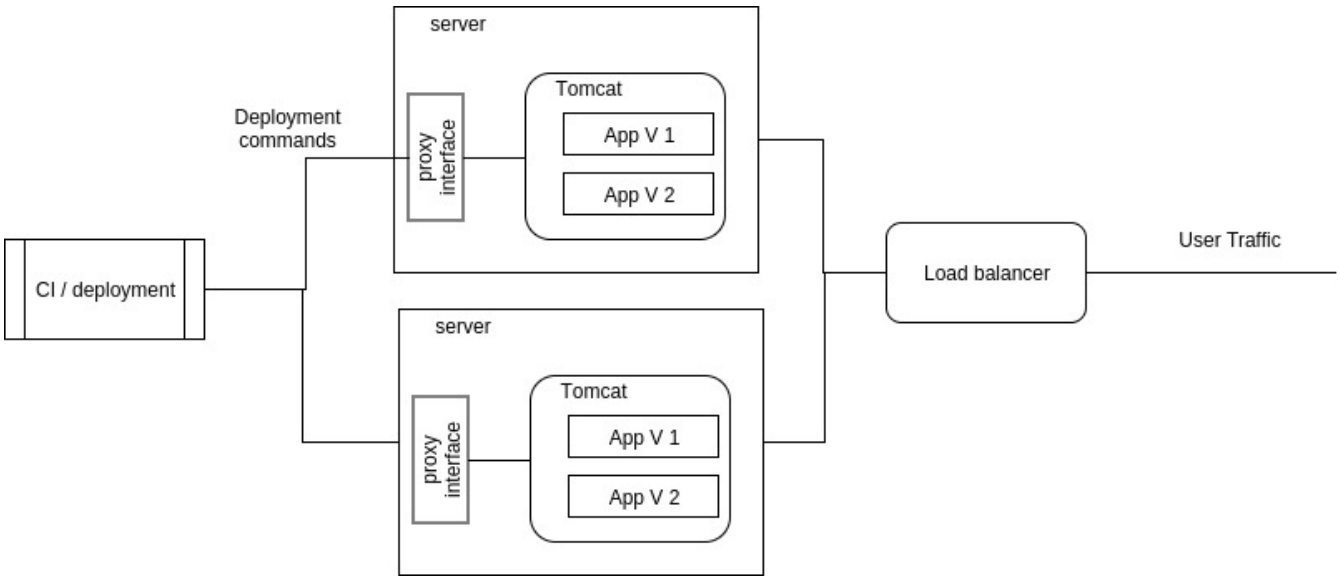

Figure 4.2 Approach 1 - Tomcat as a cluster

Figure 4.2 depicts the approach 1 as a cluster of nodes. Even though it has been displayed 2 server nodes, we have tried this with multiple nodes in the environment.

4.4.4. Drawbacks

Even though we have selected this approach as an alternative to the A/B deployment this also has few known weaknesses.

Hardware resources

Since the parallel deployment is used to keep multiple application versions up and running at a given moment, the amount of hardware resources can be consumed more than the general usage. In order to limit this behavior, multiple application versions might require undeploy accordingly.

In that case, we cannot keep the multiple application versions standby because of the over usage of hardware resources.

Application logs

Logs are critical functions for any enterprise applications. Generally, applications are not written in a way to keep separate log files for each version. In this case, multiple applications may write to the same log file since the parallel deployment is keeping multiple application contexts up and running at the same time. This may introduce performance issues as well as a discrepancy in the log files.

Tomcat Failures

In case of failure in the Tomcat server, all the application versions may be unavailable for the clients. But this can be mitigated with the use of redundant physical servers.

JVM Failures

When running the different application versions on the same Tomcat instance, it means using the same JVM throughout all the applications. If a single application or an environmental issue causes an application to crash the JVM, then all the versions may crash at the same time. Considering the availability, this may count as a major risk in this architecture if it is running on a single node.

In order to avoid the unavailability due to both Tomcat and JVM failures, it is advisable to use redundant configurations for the application nodes.

4.5. Approach 2 - Using Linux containers

Linux containers [12] can be identified as one of the rapidly expanding areas when it is about the application deployment. It has now developed into a state where it is considered as container based deployment from the initial steps as development stage. This has become much popular with its support for the microservices architecture which is also favored by the current industry.

With the support of cloud infrastructure services, container hosting as a service is also a part of their business. Hence this is something we cannot omit when considering an alternative deployment approaches.

Without completely relying on container hosting platforms such as Kubernates [12] or Amazon EC2 Container Service [1], we have adhered Docker containers as to take only the required use of the container technology by aligning to boundaries of the scope of this research.

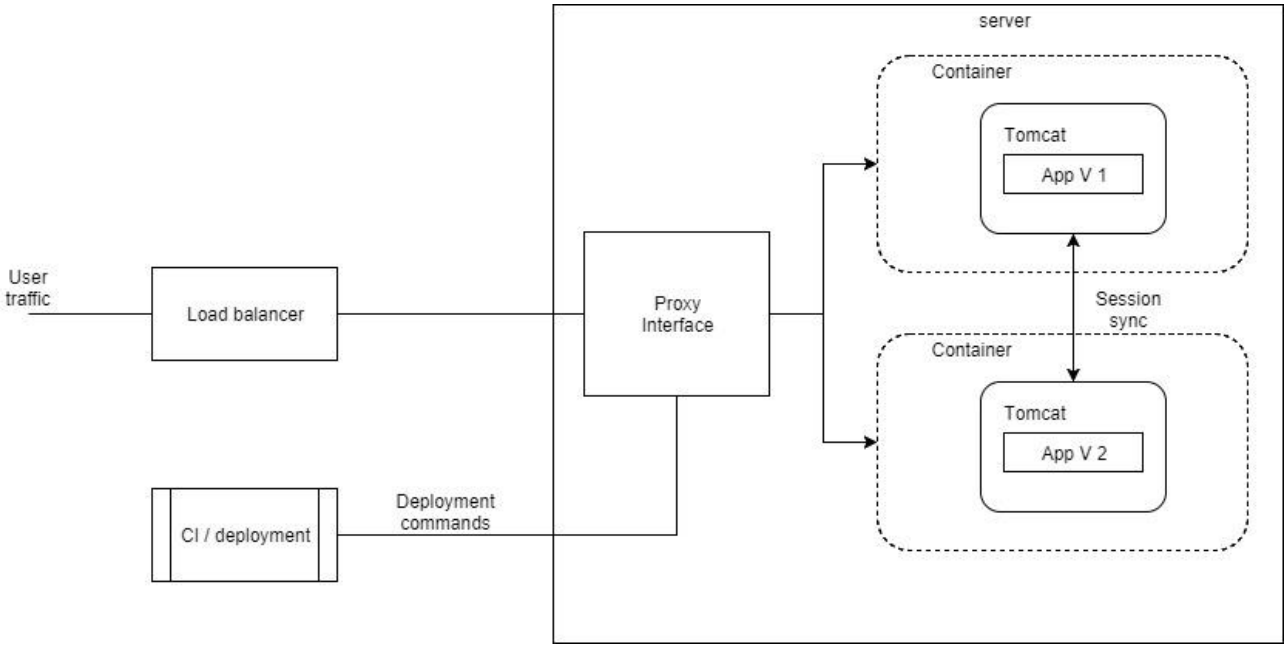

Figure 4.3 Approach 2 - Tomcat using Linux containers

Above image Figure 4.3 is displaying the approach 2 with the use of Linux containers as an alternative deployment approach.

In this scenario, each Tomcat server is embedded in a separate Linux container. In other words, for each new application version, a new container is created with a Tomcat server inside it. By this method, it overcomes some of the issues that were there in the approach 1.

In the approach 1, all the application versions are deployed inside the same Tomcat instance. This may lead to some conflicting log files, since both the versions may write into the same application log file at the same time. Addition to that, it limits the risk of failing the Tomcat instance itself due to an issue in the new application version. In the approach 1, if such an error occurs in the Tomcat instance, both the application versions may impact and may lead to downtime.

We have been able to mitigate the above issues using Linux containers with the approach 2. With this approach, the logs are kept inside the container itself, therefore writing to the same application log file will not be an issue.

4.5.1. Proxy Interface

Proxy interface is placed between the application server (docker container) and the outside load. Proxy is consist of two parts
Nginx server (the actual proxy)
Python helper app
Nginx server is configured to proxy the traffic dynamically to the appropriate docker container. Also, it directs the deployment control requests from CI/CD jobs to the python helper app.

Python helper app is a simple Python API with backend integrated into Nginx and the Docker container engine.

Upon a deployment request, this application creates a separate Docker image with the application name and the version and runs it as a separate container. Addition to that it controls Nginx proxy distribution by dynamically allocating newly created application container for the Nginx.

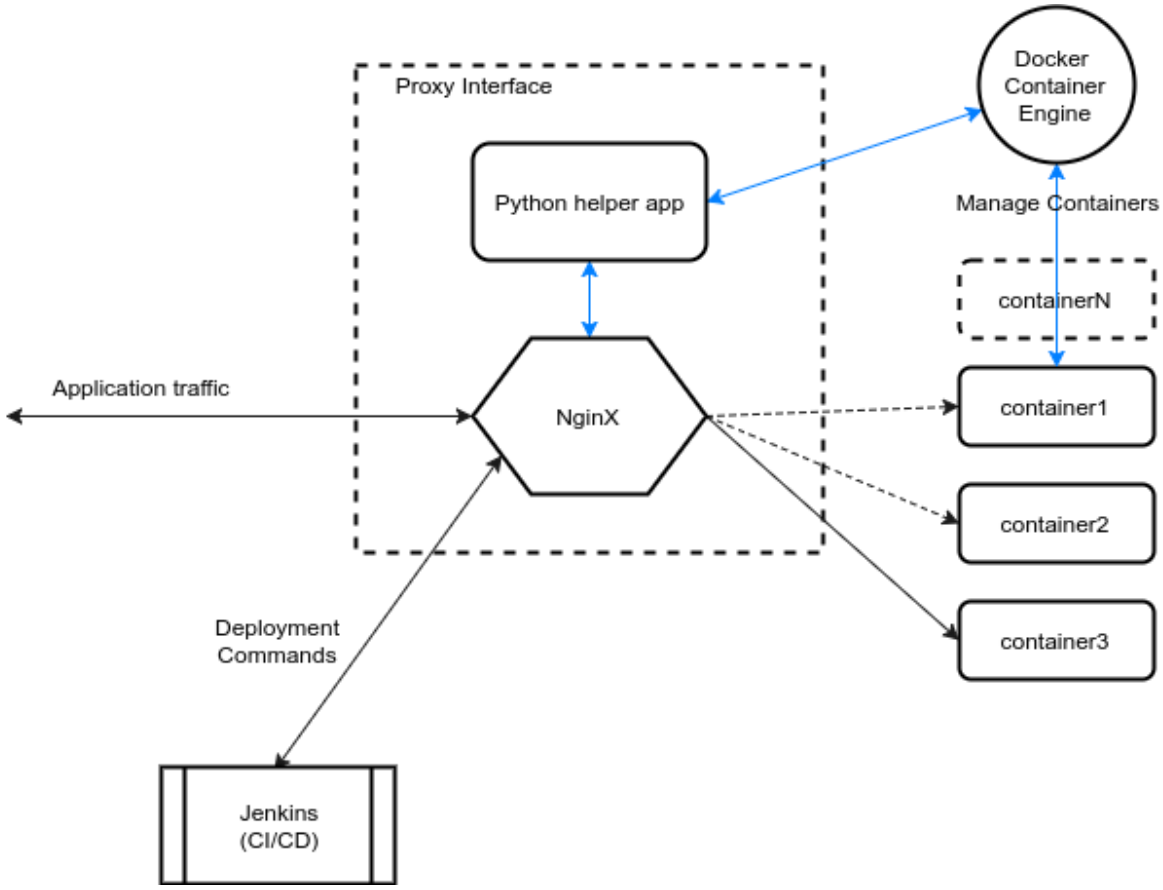

Figure 4.4 Approach 2 - Proxy interface

The proxy interface provides a set of APIs and a GUI to manage versions and deployment tasks. API can be used by CI/CD pipelines such as Jenkins.

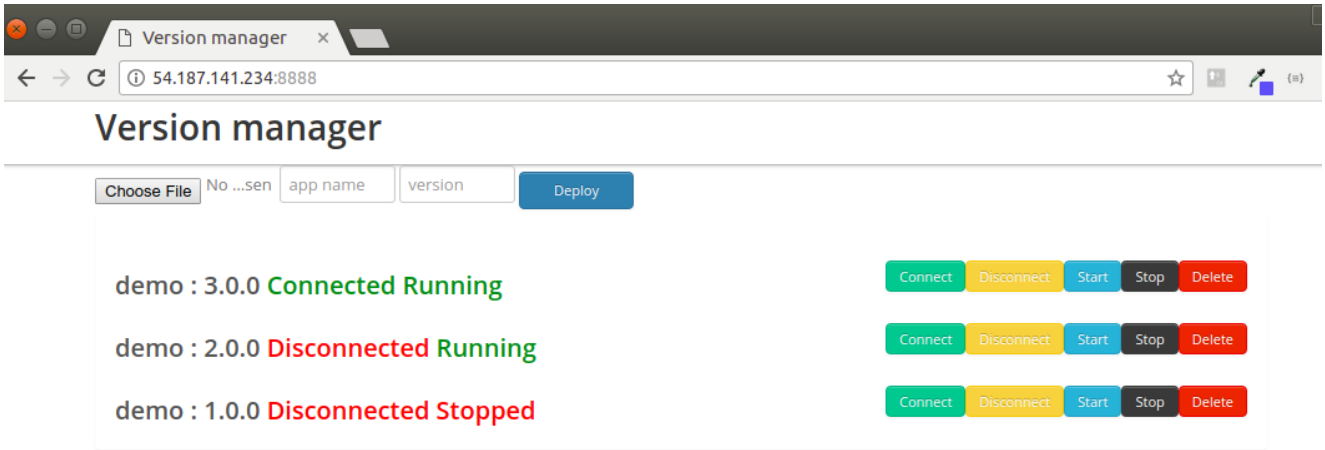

Figure 4.5 Proxy interface GUI

With the proxy interface GUI, it is capable to deploy a war file given the application name and the version. With this, it can maintain multiple versions and control which version should be up and running and connected to the live traffic.

In Figure 4.5, the app named demo has deployed as 3 subsequent versions. Older versions can be stopped and disconnected from the live traffic. As a rollback plan, the previous stable version of the app can be kept running but disconnected from the live traffic. Whenever it is required to do a rollback it can be easily connected to the live traffic and disconnect the later version.

Older versions can be deleted and removed entirely from the picture. Deploying the app is a matter of uploading the war file and the application starts up and running within seconds with the use of pre-configured Linux container images.

4.5.2. Deployment mechanism

A proxy interface has been used to communicate with each Linux container and the Tomcat server when deploying new application versions.

Same as the approach 1, the proxy interface provides a GUI as well as an API for deployment. When it is required to deploy a new application version, new application package URL was provided to the proxy interface either via GUI or API. The proxy interface will create a new Linux container with the version and deploy the application inside the Tomcat. Cut over for the new version can be controlled by the proxy interface.

Unlike in the approach 1, actual user traffic is also directed via the proxy interface. Therefore proxy can decide which version of the container it should be redirected.
One of the challenges in this approach is to keep the active sessions when deploying a new application version. This was not an issue in the approach 1 since it uses the same Tomcat instance. In order to overcome this, we had to configure each Tomcat with the application version as a cluster of instances to share the session details. Each instance is configured to sync the session information among each application version via JMX.

This mechanism helps to quickly turn off the old container instance to save up the host resources. Otherwise, both applications may utilize some amount of the host machine resources.

After the successful deployment of a package, the old container will be stopped and, it can be controlled via the proxy interface.

4.5.3. NginX

An instance of NginX server is kept as an inbuilt service inside the proxy interface. This NginX instance is specially configured to dynamically add/remove proxy backends. For that Nginx plugin has been used to dynamically assign proxy upstream [23]. Dynamic proxy upstream mechanism allows it to control the proxy by the python helper app APIs.

This feature mechanism allows the Python helper app to target the external traffic into the right application version. Apart from that, it can be also used as a communication medium between the app container and the proxy for deployment, un-deployment and health check functionalities.

The configuration file of the NginX has been attached to Appendix - I

CHAPTER 5

SYSTEM EVALUATION

5.1. Introduction

The evaluation was conducted on considering five main criteria.

Duration of the deployment process
Resources
Client impact
Cost
Rollback capability

Amazon EC2 environment was used in order to evaluate the above techniques. Using a well-established and widely used cloud infrastructure was selected considering the relevance and the validity of the evaluation methodology.

As the infrastructure, single Elastic Load Balancer (ELB) was used with connecting to 3 application server nodes.

T2.micro instance type was used in order to create simulate a scaled-down environment appropriating to the simplicity of the application used.

In AWS platform, t2.micro machine has the below configurations.

CPU: 1vCPU (Virtual CPU)
Memory: 1GB
Storage: EBS (Elastic Block Storage)

AWS platform defines T2 range instances as Burstable Performance Instances where they provides a baseline level CPU performance with the ability to burst above the baseline.

Conventional A/B deployment and the proposed approaches were evaluated against the above-mentioned criteria as below.

5.2. Load generation

While running the deployment process, the variable load was sent to the demo application in order to evaluate the user experience. In the perfect A/B scenario, the user impact should be zero for the application deployment.

Jmeter [22] application has been used in order to simulate the client usage for the demo application.

Jmeter is an industry wide recognized tool which is used for performance tests. It comes with necessary tools and configurations to generate the required load and also to observe the results.

For the evaluation, it has been used 5 versions of the same application. It is to be noted that only the version number has been changed in each of them in order to make it performance wise consistent. Therefore it is assumed that application version does not affect the performance or the deployment complexity of the process.

While the deployment process goes on Jmeter was simulated multiple user requests. Starting from 20 to 100 concurrent users. The concurrent users were able to generate 46 to 67 transactions per second (TPS).

Increasing concurrent users lead to increase the number of requests that hit the application or in other words it increases the TPS value. Still, the TPS value can depend on a number of other parameters as well. Eg: Network traffic during the time of the test.

5.3. A/B deployment evaluation

A typical A/B deployment process was created as a sequential Jenkins jobs.

Figure 5.1 A/B deployment Jenkins jobs

Each Jenkins job was to conduct a separate task for the deployment process.

1. Build the app and push the package to Amazon S3
2. Spin-up new servers for the application version
3. Check app node state and verify the application is running
4. Add the new nodes to load balancer and remove old nodes

Hardware resource monitoring has not considered in the A/B deployment since isolated application node is allocated to each application version. Therefore hardware resources are not changing over application deployment.

5.4. Evaluation alternative approaches

As mentioned above, the same type of environment and infrastructure were used to evaluate the approaches.

Approach 1: parallel deployment with Tomcat

Approach 2: using Linux containers

All the approaches were executed 4 times with 5 application versions while generating a load for the application using variable TPS values.

Unlike A/B method, with this approach, we are trying to use the same application node for multiple versions. Therefore we have to monitor the hardware resource usage as well.

5.5. Results and discussion

It is observed that A/B deployment was taking longer duration for the entire deployment process. The average time of duration was 112 seconds.

Figure 5.2 A/B deployment duration

5.5.1. A/B deployment deployment with 20 concurrent users

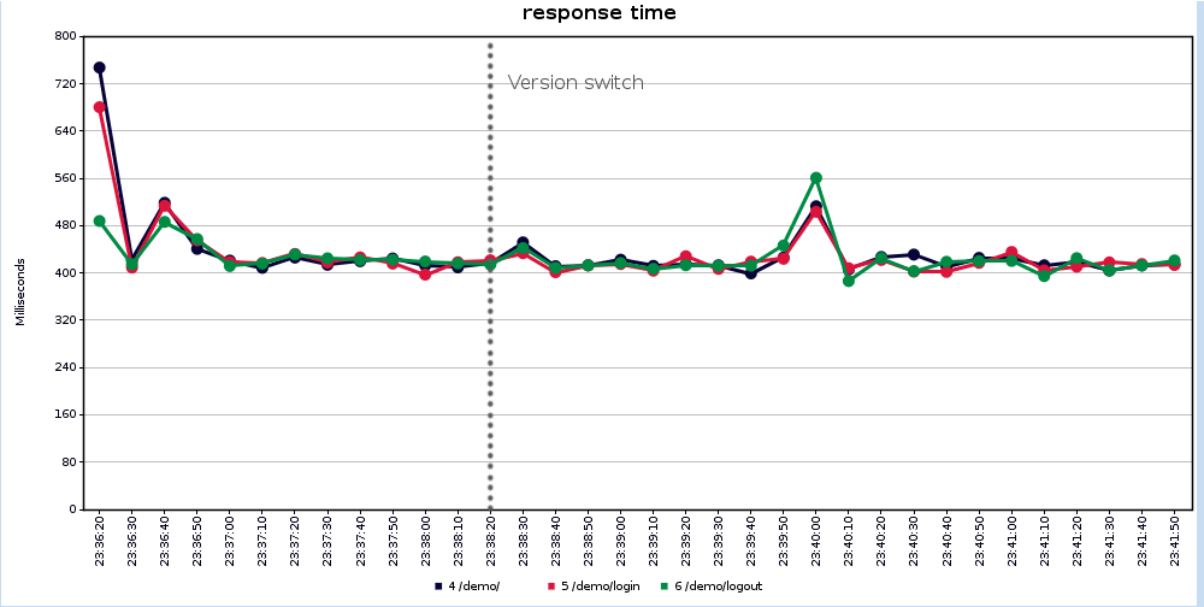

Figure 5.3 A/B deployment - Response time for 20 concurrent users

The load was generated with 20 concurrent users lead to reach the application with 46 TPS throughput.

Table 5.1 A/B deployment - Load summary table for 20 concurrent users

Label	# Samp...	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throug...	Receive...	Sent K...
4 /demo/	5396	426	348	681	715	862	289	2839	0.00%	15.6/sec	7.92	5.52
5 /dem...	5394	424	349	682	721	837	298	2631	0.00%	15.7/sec	5.08	8.24
6 /dem...	5387	423	349	681	720	825	298	2658	0.00%	15.7/sec	7.92	6.88
TOTAL	16177	425	349	681	720	838	289	2839	0.00%	46.6/sec	20.77	20.48

According to the figure 5.3, a small number of concurrent users, the response time does not get impacted much by A/B deployment.

Here it is to be noted that ELB is taking care of draining the connection from B nodes without immediately disconnecting them from traffic.

No Errors has been returned to the clients.

5.5.2. A/B deployment deployment with 100 concurrent users

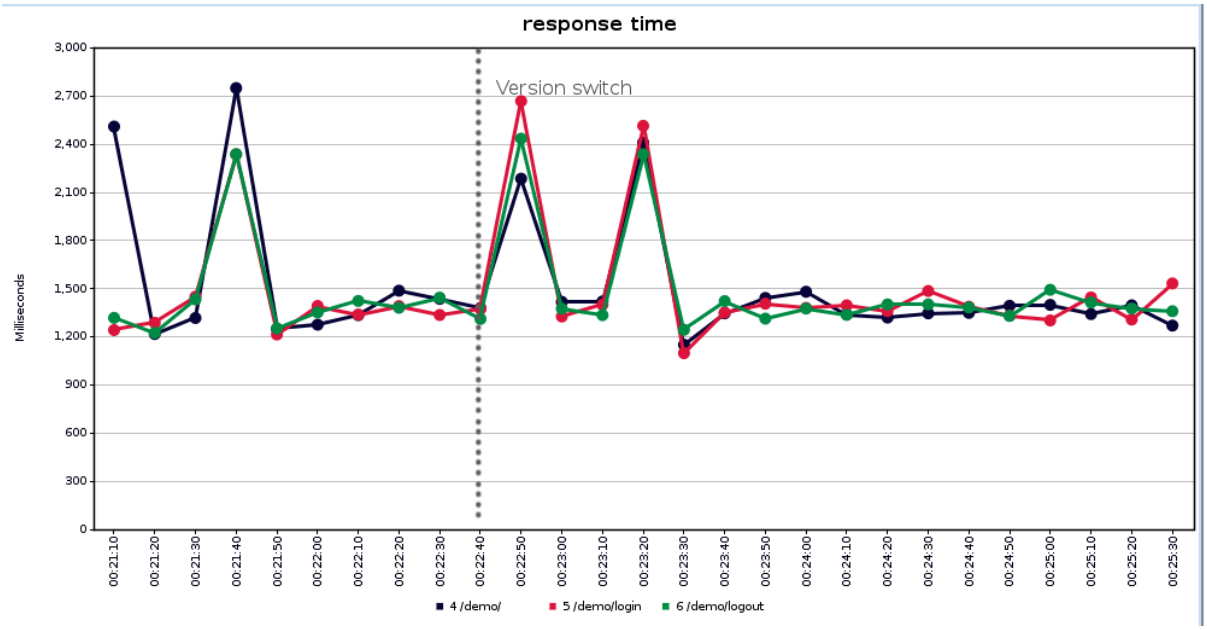

Figure 5.4 A/B deployment - Response time for 100 concurrent users

The load was generated with 100 concurrent users lead to reach the application with 67 TPS throughput.

Table 5.2 A/B deployment - Load summary table for 100 concurrent users

Label	# Sam...	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throug...	Receive...	Sent KB...
4 /demo/	6127	1467	1044	3086	3791	7062	298	12725	0.00%	22.8/sec	11.58	8.07
5 /dem...	6086	1449	1045	3018	3698	7136	327	12787	0.00%	22.7/sec	7.36	11.95
6 /dem...	6052	1444	1049	2961	3717	6660	301	13086	0.00%	22.7/sec	11.44	9.94
TOTAL	18265	1453	1046	3029	3738	6944	298	13086	0.00%	67.8/sec	30.22	29.79

Similar to the previous test it has been observed to achieve 100% success rate for the client requests.

Response time has been quite disturbed when compared with small amount of concurrent users.

5.5.1. Approach 1: using parallel deployment with 20 concurrent users

Jmeter has been generating load with 20 concurrent users and 46 TPS has been reached to the application.

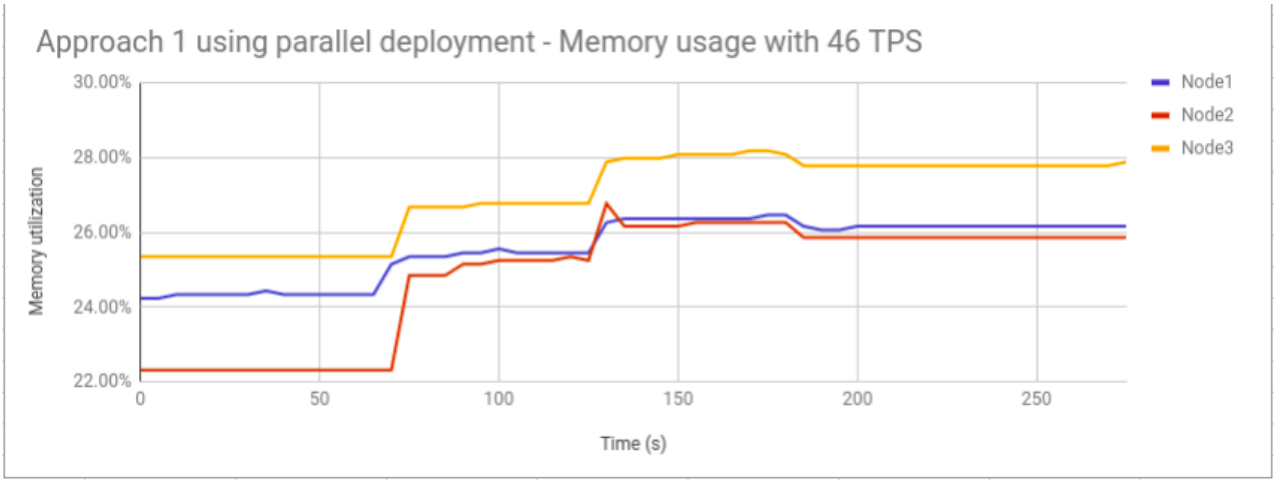

Figure 5.5 Approach 1 using parallel deployment - memory usage with 20 concurrent users

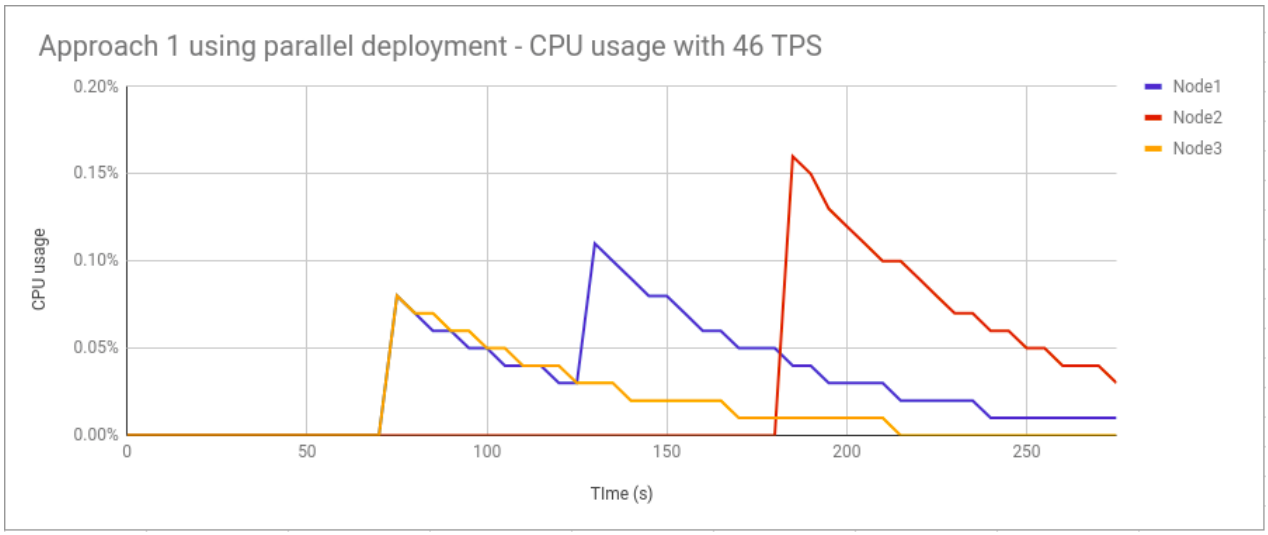

Figure 5.6 Parallel deployment - CPU usage with 20 concurrent users

Table 5.3 Parallel deployment - Load summary table for 20 concurrent users

Label	# Samp...	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throug...	Receive...	Sent KB...
4 /demo/	3329	430	342	680	720	869	289	4811	0.00%	15.4/sec	7.85	5.47
5 /dem...	3323	436	348	680	718	877	299	7838	0.00%	15.6/sec	5.06	8.21
6 /dem...	3316	417	344	677	710	851	299	3699	0.00%	15.8/sec	7.96	6.91
TOTAL	9968	428	345	680	717	864	289	7838	0.00%	46.1/sec	20.54	20.26

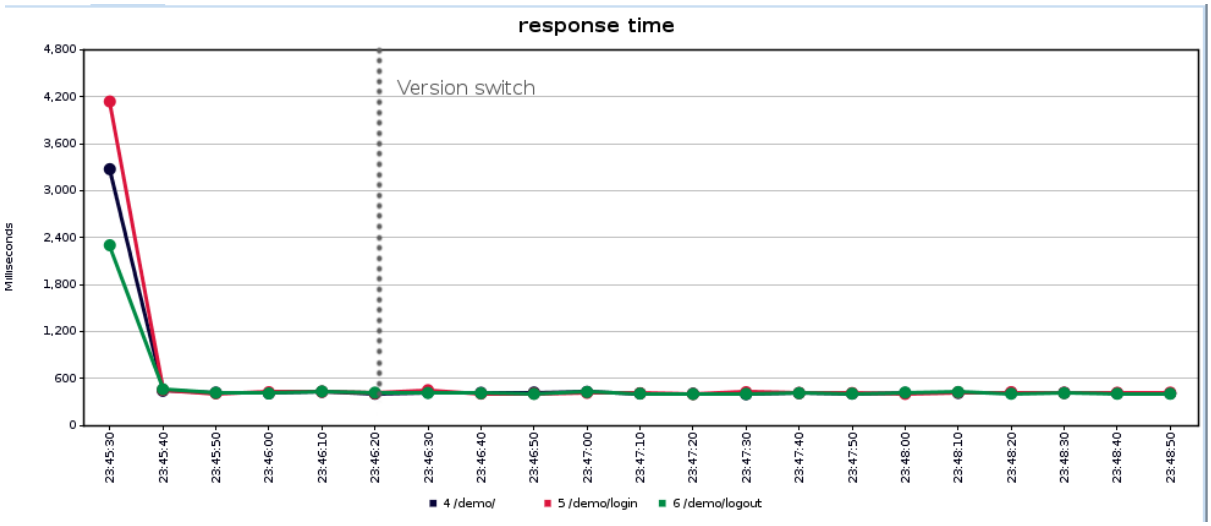

Figure 5.7 Approach 1 using parallel deployment - Response time for 20 concurrent users

It is to be noted that memory usage displays significant behavior to not release the memory after each deployment. This is because Tomcat requires a fair amount of memory for keeping parallel application versions available. Addition to that we don't have the ability to observe how many users are still using connected to the old version of the application. Therefore it has to be manually un-deployed in order to free up the memory.

Similar to earlier, 0% error rate was observed. CPU spikes were observed at the time of deployment but later, the usage has been normalized.

It could be observed a steady response time throughout the deployment process.

5.5.2. Approach 1: using parallel deployment with 100 concurrent users

Jmeter has been generating load with 100 concurrent users and 53 TPS has been reached to the application.

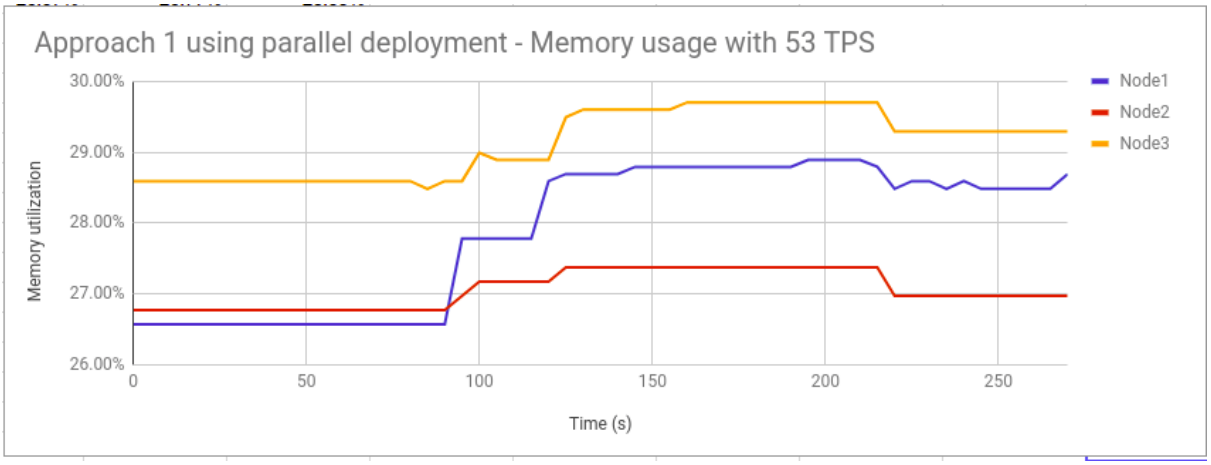

Figure 5.8 Approach 1 using parallel deployment - memory usage with 100 concurrent users

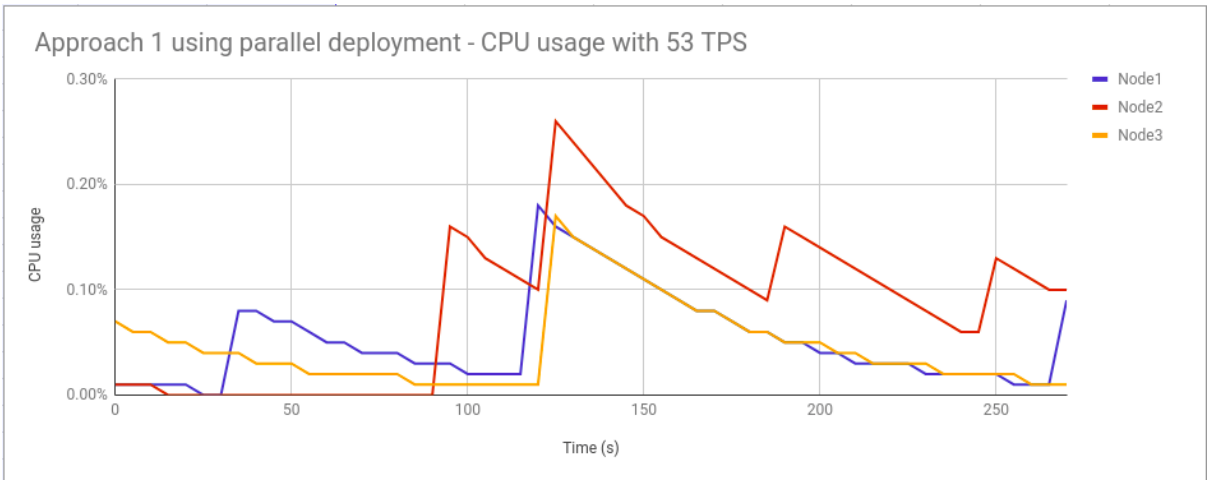

Figure 5.9 Parallel deployment - CPU usage with 100 concurrent users

Table 5.4 Parallel deployment Load summary with 100 concurrent users

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throughput	Received ...	Sent KB/sec
4 /demo/	11403	1842	1137	3892	5370	8798	311	20134	0.00%	17.8/sec	9.04	6.30
5 /demo/l...	11371	1889	1149	3980	5549	9387	308	23098	0.00%	17.7/sec	5.75	9.34
6 /demo/l...	11339	1873	1150	3939	5409	9096	309	20137	0.00%	17.7/sec	8.92	7.75
TOTAL	34113	1868	1144	3939	5441	9080	308	23098	0.00%	53.2/sec	23.69	23.36

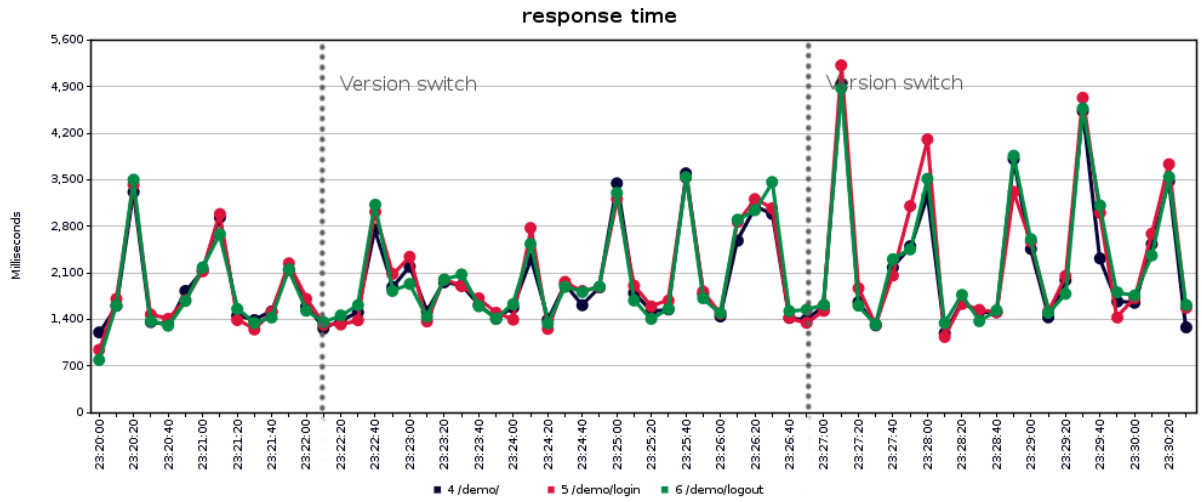

Figure 5.10 Parallel deployment - Response time for 100 concurrent users

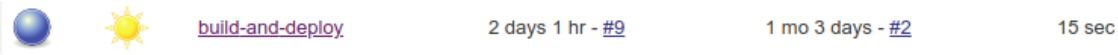

Figure 5.11 Parallel deployment duration

100 concurrent users load has been used to this test and it has reached up to 53TPS. Similar to the earlier scenario, it can be observed that Tomcat is holding the memory for each app version without releasing it back.

Response time has been observed to go all over the place without a steady line as earlier. It suggests that parallel deployment approach is unable to achieve a steady response time over the increase of usage.

Average deployment duration was observed to be smaller as 19 seconds.

5.5.2. Approach 2: using Linux containers with 20 concurrent users

Jmeter has been generating load with 20 concurrent users and 47 TPS has been reached to the application.

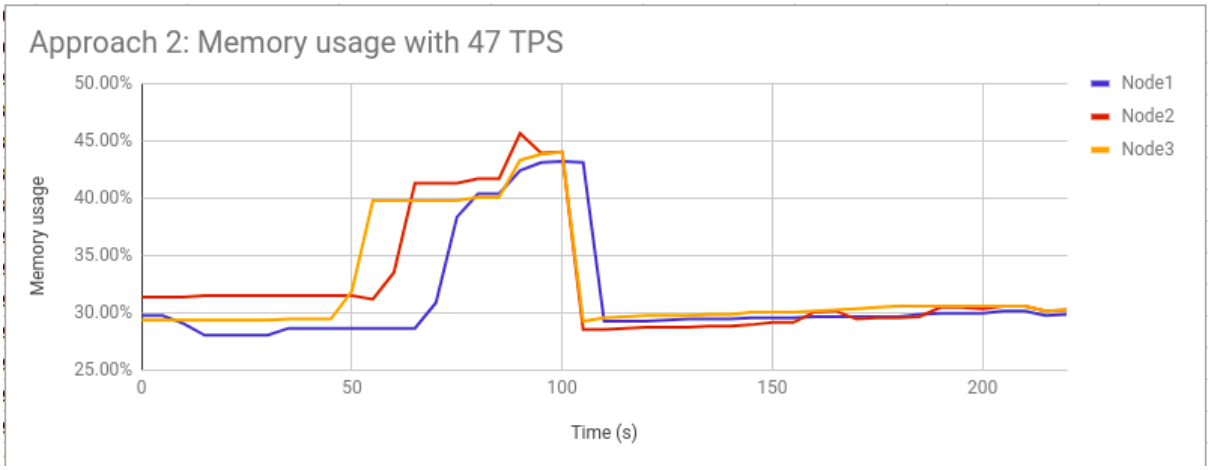

Figure 5.12 Linux containers - Memory usage with 20 concurrent users

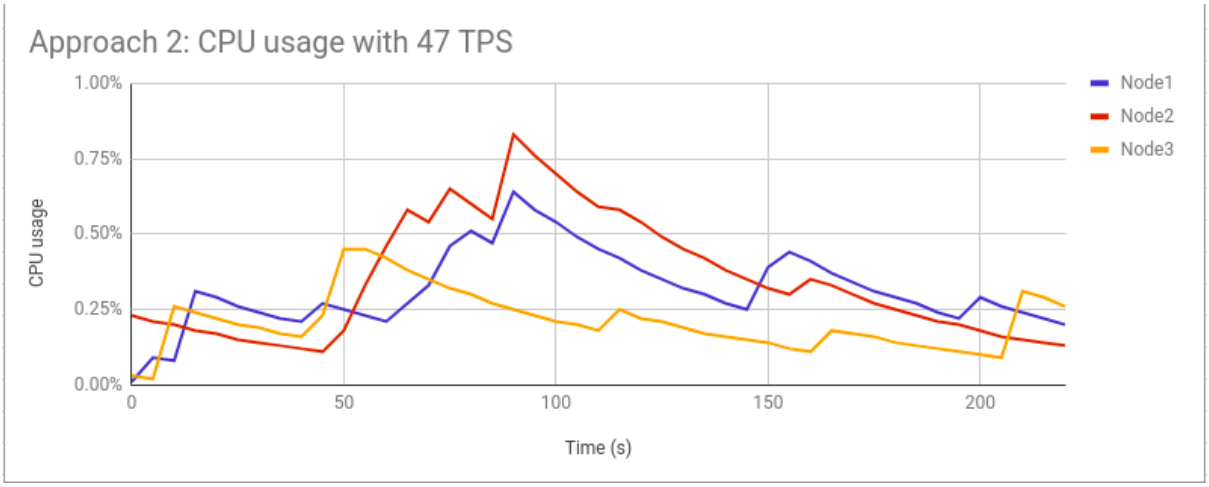

Figure 5.13 Approach 2 using Linux containers CPU usage with 20 concurrent users

Table 5.5 Linux containers - Load summary table for 20 concurrent users

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throughput	Received KB...	Sent KB/sec
4 /demo/	7160	428	347	679	720	820	288	10042	0.00%	15.8/sec	8.36	5.59
5 /demo/login	7155	419	342	679	716	820	298	9009	0.00%	15.9/sec	5.49	8.36
6 /demo/log...	7149	417	349	680	713	812	298	1478	0.00%	16.0/sec	8.40	7.00
TOTAL	21464	421	348	680	716	820	288	10042	0.00%	47.3/sec	22.08	20.77

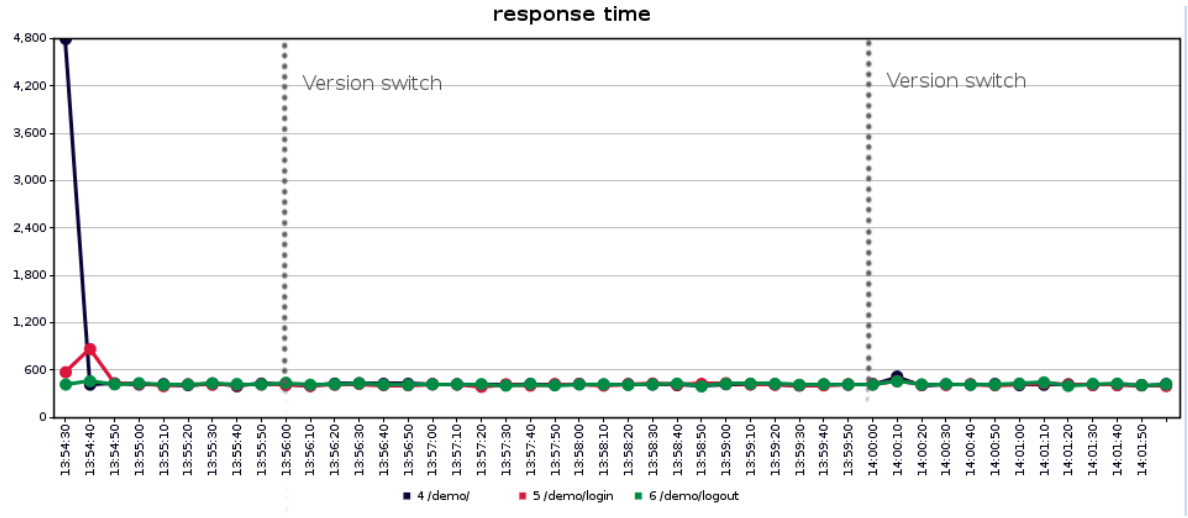

Figure 5.14 Linux containers - Response time for 20 concurrent users

As usual, no client errors were observed for any of the requests. It was observed a steady and low response time with a really small spike for the deployment. Also, it was observed spikes in CPU and memory usage at the time of deployment but then it has been lowered back to normal usage quickly.

5.5.2. Approach 2: using Linux containers with 100 concurrent users

Jmeter has been generating load with 100 concurrent users and 47 TPS has been reached to the application.

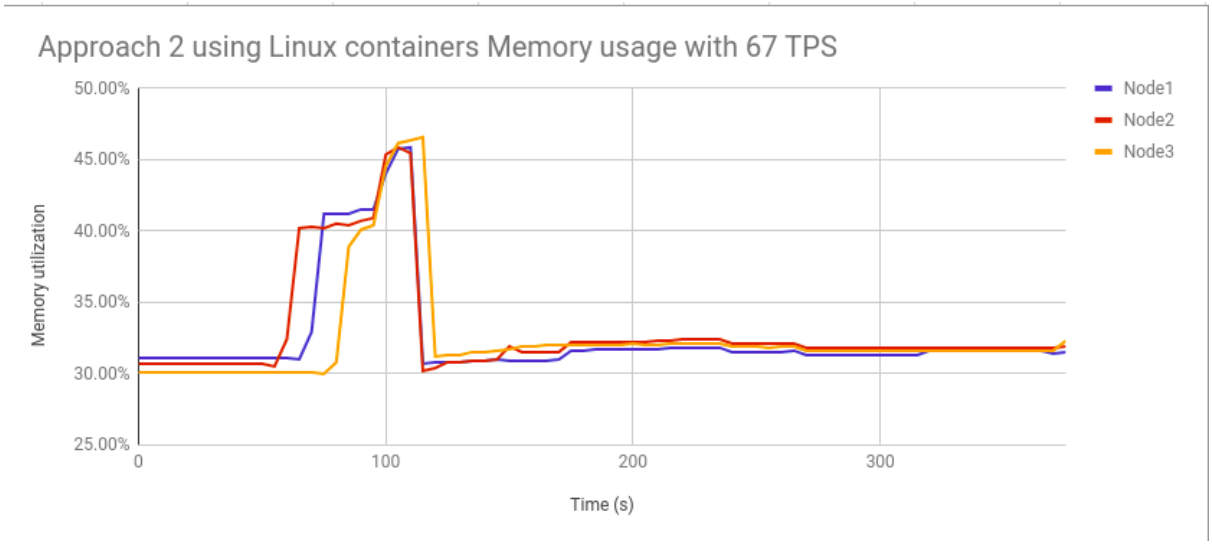

Figure 5.15 Linux containers Memory usage with 100 concurrent users

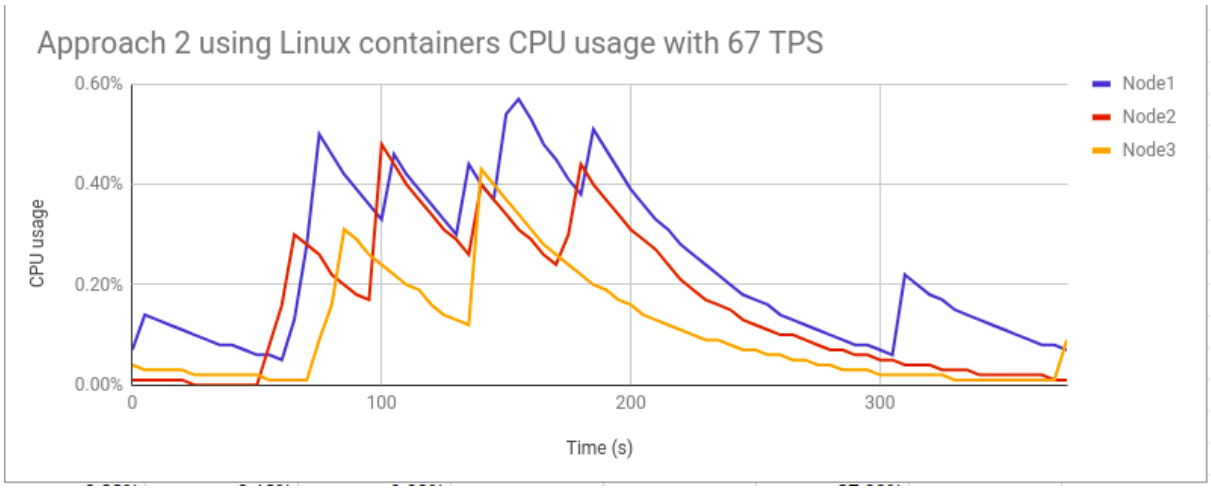

Figure 5.16 Linux containers CPU usage with 100 concurrent users

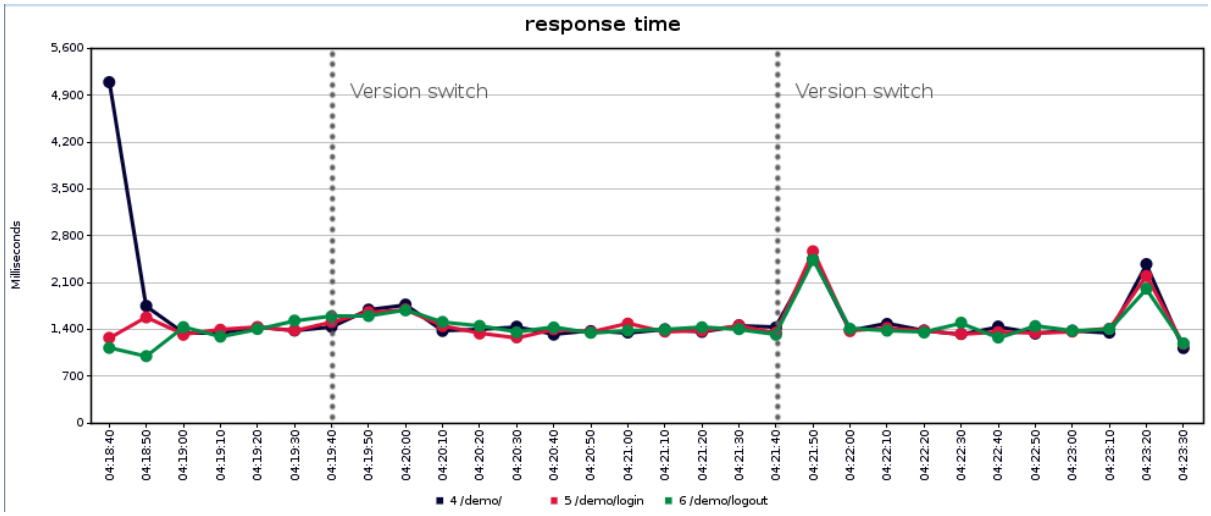

Figure 5.17 Linux containers - Response time for 100 concurrent users

Table 5.6 Linux containers - Load summary table for 100 concurrent users

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throughput	Received ...	Sent KB/s...
4 /demo/	6679	1495	1041	3089	3810	6721	322	11409	0.00%	22.6/sec	11.99	8.02
5 /demo/l...	6644	1451	1045	3017	3689	5892	319	11895	0.00%	22.7/sec	7.86	11.96
6 /demo/l...	6607	1444	1043	3029	3720	5390	312	11647	0.00%	22.8/sec	12.00	9.99
TOTAL	19930	1464	1043	3049	3735	5927	312	11895	0.00%	67.5/sec	31.52	29.64

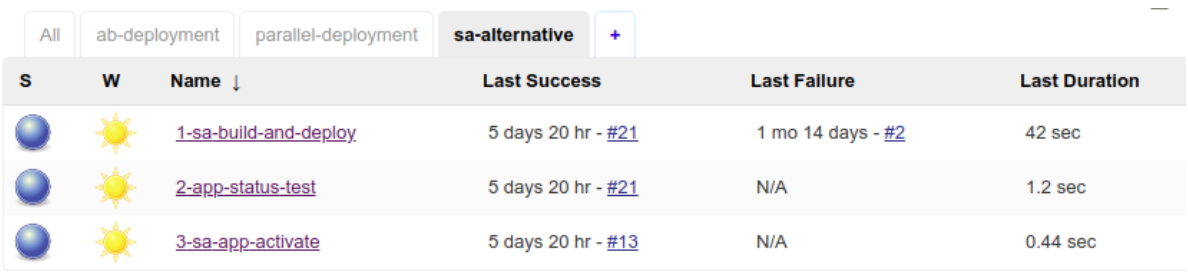

Figure 5.18 Linux containers deployment duration

Approach 2 deployment using Linux containers was observed to have smaller deployment duration such as an average of 40 seconds.

Similar as earlier, the deployment processes exhibited 0% error rate for client requests.

Same as the earlier scenario, memory and CPU spikes were observed during the deployment but they have been returned to normal usage quickly.

Small spike has observed in the response time graph (Figure 5.17) but comparatively the deviation is smaller with respect to both A/B approach and parallel deployment approach.

CHAPTER 6

CONCLUSION

6.1. Conclusion

It can be concluded that alternative deployment approaches could be used instead of A/B deployment depending on the application usage and requirements.

None of the approaches were made any significant impact on the client requests same as A/B approach and all of them were able to serve with 0% error rate.

6.1.1. Alternative approach 1: Parallel deployment using Tomcat

This approach can be recommended for low usage applications since the deployment goes performs smoothly with lower concurrent users. Additionally, this approach could be accommodated easily by legacy applications where they already use Tomcat as the underlying container.

Cost wise this can be said to be cheaper when compared with the A/B approach since a single set of servers can be used over multiple application versions.

Addition to that this can be concluded as the deployment process with the smallest duration. This smaller duration has been achieved because the server itself supports this behavior inherently.

It can be noted few drawbacks in this where it comes to separating application versions. Because the server itself is responsible to exposing each application version to the traffic, we have little control over the versioning from the outside.

Also when it comes to the resources, it can be concluded this as a memory usage heavy approach. Hence, it is to be expected the host server to have enough memory available to keep multiple application versions.

6.1.2. Alternative approach 2: Using Linux containers

This approach can be recommended for the applications where both cost and performance are significant requirements. Cost wise this approach also cheaper compared with A/B deployment since only a single set of servers are required.

No significant usage of hardware resources was detected because soon as the new application version is ready running, the proxy is taking care of stopping the old application version. Therefore both application versions are running only for few seconds. However, it is expected to have a host machine with enough hardware resources to support two application versions as the worst case scenario.

This approach can be recommended for the applications where it requires to frequently switch throughout multiple application versions.
For example: services in microservices architecture.
Applications with seasonal features in different versions.

Additionally, it can be suggested that proposed alternative approach is much appropriate for the applications that are required to run on single stand alone nodes in a highly integrated environment. In this case, approach 2 is more capable of handing over features that are facilitated by A/B deployment approach.

6.1.3. Cumulative conclusions of both alternative approaches

Since both exhibited 100% success rate for all the user requests, these approaches successfully meet objectives of the research specific to high availability. Also the cost effectiveness can be highlighted in both the approaches.

Both approaches can be recommended for small and middle size applications specially where it requires to stay in a single stand alone server in a legacy integrated environment.

6.2. Study limitations

In this research, it has been considered only Java servlet based application deployment process as a proof of concept. It can be anticipated other technologies can also adopt the proposed architecture and the deployment processes in similar ways.

For the evaluation of the deployment processes it has been used AWS environment and therefore it has been evaluation criteria has been limited to the scope of AWS cloud environment.

Since it has been used AWS environment for the evaluation, it is assumed that the results of the evaluation could be limited to the AWS cloud environment considering the inherent additional features that it provides for the infrastructure.

The sample application which was used as the system under test has been developed as a simple Java servlet web application. The results are based on the performance of the sample application and it is assumed the results may vary according to the nature and the complexity of the application.

Jenkins automation server has been used in order to test the deployment processes with continuous integration. Therefore the evaluation of the deployment methods has been limited to Jenkins integration.

6.3. Future works

Since this research only has been concerned on Java web applications in Tomcat as the server, it can be further extended to other platforms as well.

Additionally the approach 2 does not concern about the security of the APIs. It is assumed the APIs are not exposed to outside of the firewall. Therefore as future work a security layer can be implemented into the proxy interface.

Parallel deployment is currently available in newer version of Tomcat as an advanced feature. It rate to observe this is being used practically in the industry. Similar capacity of features can be also introduced into other web servers also covering common platforms. If the server platform itself was able to support this idea, it will be much easier for the deployment engineering process in future.

Subject area of virtualization with Linux container is noticed to be growing faster in past few years. It is to be expected to grow this area more with continuously growing cloud platforms.

REFERENCES

[1] John, G., Mui, A., Vlasceanu, V., 2016. Blue/Green Deployments on AWS (White paper). Amazon Web Services, Inc.

[2] Maxim, T., 2015. Reducing Downtime during Software Deployment. Tampere University of technology.

[3] Vincent, E., Brandic, I., Maurer, M., Breskovic, I., n.d. SLA-Aware Application Deployment and Resource Allocation in Clouds. Presented at the 2011 35th IEEE Annual Computer Software and Applications Conference Workshops, IEEE

[4] Cheng, F.-T., Wu, S.-L., Tsai, P.-Y., Chung, Y.-T., Haw-Ching, Y., 2005. Application Cluster Service Scheme for Near-Zero-Downtime Services. Presented at the Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference, IEEE. doi:10.1109/ROBOT.2005.1570743

[5] Sun, R., Yang, J., He, Z., 2013. An Approach to Minimizing Downtime Induced by Taking Live Snapshot of Virtual Cluster. Presented at the 2013 International Conference on Cloud and Service Computing, IEEE. doi:10.1109/CSC.2013.18

[6] Cornell, D., LarsFeldmann, L., Afanasjev, A., Dimitrov, E., 2005. Method and Apparatus for Facilitating Deployment of Software Applications with Minimum System Downtime. US 20050257216A1.

[7] Rajamani, K., Viswanathan, G., Li, W., Iyer, C., 2005. Minimizing Downtime for Application Changes in Database Systems. US 2005/0251523 A1.

[8] Ponemon Institute, 2016. Cost of Data Center Outages. Available at: http://datacenterfrontier.com/white-paper/cost-data-center-outages/.

[9] Jez Humble, D.F., 2010. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation 1st Edition. Pearson Education, Inc.

[10] M. Miglierina, "Application Deployment and Management in the Cloud," presented at the 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, 2014.

[11] Apache Tomcat® 8.0.x 2017, Computer Program, Apache Software Foundation, http://tomcat.apache.org/tomcat-8.0-doc/.

[12] D. Rajdeep, R. A Reddy, and K. Dharmesh, "Virtualization vs Containerization to Support PaaS," in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, Boston, MA, USA, 2014.

[13] R. David K, *Kubernetes - Scheduling the Future at Cloud Scale*. CA USA: O'Reilly and Associates

.

[14] C. Damodar, *Tomcat 6 Developer's Guide*. MUMBAI: Packt Publishing.

[15] "NGINX," *NGINX Wiki*, 2017. [Online]. Available: https://www.nginx.com/resources/wiki/.

[16] V. Moreno, M. R. S, H. E., and L. I. M, "Orchestrating the Deployment of High Availability Services on Multi-zone and Multi-cloud Scenarios," *Journal of Grid Computing*, no. Issue 1/2018, 2017.

[17] W. Timothy, C. Emmanuel, R. K. K., S. Prashant, van der M. Jacobus, and V. Arun, "Disaster recovery as a cloud service: economic benefits & deployment challenges," in *HotCloud'10 Proceedings of the 2nd USENIX conference*, Boston, MA, 2010, pp. 8–8.

[18] X. Wenfeng, Z. Peng, and W. Yonggang, "A Survey on Data Center Networking (DCN): Infrastructure and Operations," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 1, pp. 640–656, Nov. 2016.

[19] L. Eero, I. Juha, and L. Casper, "Problems, causes and solutions when adopting continuous delivery—A systematic literature review," *Information and Software Technology*, vol. Volume 82, pp. 55–79, Feb. 2017.

[20] V. Mateljan, D. Cisic, and D. Ogrizovic, "Cloud Database-as-a-Service (DaaS) - ROI," in *MIPRO, 2010*, Opatija, Croatia, 2010.

[21] H. Ines, M. Marouen, and L. Wajdi, "Cloud Service Delivery across Multiple Cloud Platforms," in *Services Computing (SCC)*, Washington, DC, USA, 2011.

[22] Apache Foundation, "JMeter," *JMeter User's Manual*, Jan-2018. [Online]. Available: http://jmeter.apache.org/usermanual/index.html.

[23] cubicdaiya, walf443, igorastds, and flygoast, *ngx_dynamic_upstream*. 2016. Available: https://github.com/cubicdaiya/ngx_dynamic_upstream.

[24] L. Min-Gu and J. Theresa L, "An Empirical Study of Software Maintenance of a Web-based Java Application," in *21st IEEE International Conference on Software Maintenance*, Budapest, Hungary, 2005.

[25] R. Sebastian and S. Andreas, "The Impact of Context on Continuous Delivery - Full-scale Software Engineering / Current Trends in Release Engineering," RWTH Aachen University, Germany, 2016.

[26] D. Alan, “Software Deployment, Past, Present and Future,” in *Future of Software Engineering, 2007. FOSE ’07*, Minneapolis, MN, USA, 2007.

[27] Gideon and D. Ewa, “Automating Application Deployment in Infrastructure Clouds,” in *Cloud Computing Technology and Science*, Athens, Greece, 2011.

Appendix I - NginX configurations

```
events {
}

http {
   include       mime.types;

         ######python interface for deployment
   server {
      listen       8888;
         # set client body size to 50M #
                  client_max_body_size 50M;
                  proxy_connect_timeout       1000;
                  proxy_send_timeout          1000;
                  proxy_read_timeout          1000;
                  send_timeout                1000;

      location / {
         proxy_pass http://depinterface;
      }

   }

         upstream depinterface {
                  server localhost:5000;
         }

         ######default backends
         upstream backends {
                  zone backends 1m;
                  server 127.0.0.1:8999;
         }

         server {
                  listen 10000;
                  # dynamic upstream config api
                  location /dynamic {
                           allow 127.0.0.1;
                           deny all;
                     dynamic_upstream;
                  }
         }

         server {
                  listen   8080;

      location / {
         # proxy for dynamic upstreams (backends)
```

```
                proxy_pass http://backends;
        }
            }
}
```

Appendix II - Python helper app source code

#app.py

```
from flask import Flask, request, send_from_directory, jsonify, redirect, url_for
import os
import requests
import docker
import json
from werkzeug.utils import secure_filename
from distutils.dir_util import copy_tree, remove_tree

app = Flask(__name__)


@app.route("/")
def root():
    return send_from_directory('ui', 'index.html')


@app.route("/index")
def index():
    return send_from_directory('ui', 'index.html')


@app.route('/<path:path>')
def send_ui(path):
    return send_from_directory('ui', path)


@app.route('/ui/deploy', methods=['POST'])
def deploy_file_from_ui():
    if request.method == 'POST':
        file = request.files['file']
        if not request.form.get('version'):
            return "version is empty"
        if not request.form.get('appname'):
            return "app package name is empty"
        version = request.form.get('version')
        appname = request.form.get('appname')
        create_workspace(appname, version)
        copy_assets(appname, version)
        filename = secure_filename(file.filename)
```

```
        # saving file to workspace
        file.save(os.path.join('workspace/' + appname + '/' + version, filename))
        # create docker file
        create_docker_file(appname, version)
        # get an available port
        port = get_available_port()
        # reserve port
        reserve_app_port(appname, version, port)
        # build image
        build_docker_image(appname, version)
        # clean workspace
        clean_assets(appname, version)
        run_container(appname, version, port)
        add_to_proxy(port)
        return redirect(url_for('index'))


@app.route('/api/activate', methods=['GET'])
def activate():
    if request.method == 'GET':
        if not request.args.get('version'):
            return "version is empty"
        if not request.args.get('appname'):
            return "app package name is empty"
        version = request.args.get('version')
        appname = request.args.get('appname')

        # start and connect app to proxy
        port = get_port_for_stopped_app(appname, version)['port']
        if not is_running(appname, version):
            print("Starting application:" + appname + ":" + version)
            run_container(appname, version, port)
        up("127.0.0.1:" + port)
        # down other applications
        applications = get_all_apps()
        print("Disconnecting other applications")
        for application in applications:
            if (application['appname'] != appname) or (application['version'] != version):
                down("127.0.0.1:" + application['port'])
        return "activated " + appname + " " + version


@app.route('/api/backends')
def backends():
```

```
    return jsonify(get_all_apps())


@app.route('/api/apps/<appname>/<version>/status')
def get_app_status(appname, version):
    if not request.args.get('health_route'):
        return "health_route is empty"
    health_route = request.args.get('health_route')
    port = get_port_for_stopped_app(appname, version)['port']
    # send a get request to the app's health route, timeout for 3 seconds
    try:
        r = requests.get('http://127.0.0.1:' + port + health_route, verify=False, timeout=3)
    except requests.exceptions.ConnectionError:
        return jsonify({'status': 'down', 'code': 500, 'err': 'Connection refused'})
    if r.status_code == 200:
        return jsonify({'status': 'up', 'code': r.status_code})
    else:
        return jsonify({'status': 'down', 'code': r.status_code})


# retrieve all available apps
def get_all_apps():
    apps = []
    client = docker.from_env()
    images = client.images.list(all=True)
    j = 0
    while j < len(images):
        image = images[j]
        if len(image.tags):
            for tag in image.tags:
                tag_parts = tag.split(':')
                appname = tag_parts[0]
                version = tag_parts[1]
                if appname != 'ubuntu':
                    running = is_running(tag_parts[0], tag_parts[1])
                    connected = is_connected(tag_parts[0], tag_parts[1])
                    apps.append({"appname": appname,
                                 "version": version,
                                 "port": get_port_for_stopped_app(appname, version)['port'],
                                 "isConnected": connected,
                                 "isRunning": running})
        j = j + 1
    return apps
```

```
def is_connected(appname, version):
    proxy_backends = get_all_proxy_connections()
    for backend in proxy_backends:
        if backend['appname'] == appname and backend['version'] == version:
            return backend['isConnected']
    return False


def does_connection_exists(port):
    r = requests.get('http://127.0.0.1:10000/dynamic?upstream=backends&verbose=')
    lines = r.text.split('\n')
    i = 0
    while i < len(lines):
        if len(lines[i]) > 0:
            line_parts = lines[i].split(' ')
            backend_parts = line_parts[1].split(':')
            if port == backend_parts[1]:
                return True
        i = i + 1
    return False


# fetch backends from nginx proxy
def get_all_proxy_connections():
    r = requests.get('http://127.0.0.1:10000/dynamic?upstream=backends&verbose=')
    lines = r.text.split('\n')
    i = 0
    backends = []
    while i < len(lines):
        if len(lines[i]) > 0:
            line_parts = lines[i].split(' ')
            backend_parts = line_parts[1].split(':')
            port = backend_parts[1]
            status = line_parts[len(line_parts) - 1]
            if status == "down;":
                connected = False
            else:
                connected = True
            app_details = get_app_for_port(port)
            backends.append({"appname": app_details.get('appname'),
                             "version": app_details.get('version'),
                             "port": port,
                             "isConnected": connected})
```

```
        i = i + 1
    return backends


@app.route('/api/backends/<application>/down')
def down(application):
    r = requests.get('http://127.0.0.1:10000/dynamic?upstream=backends&server=' +
application + '&down=')
    return r.text


@app.route('/api/backends/<application>/up')
def up(application):
    port = application.split(':')[1]
    if does_connection_exists(port):
        r = requests.get('http://127.0.0.1:10000/dynamic?upstream=backends&server=' +
application + '&up=')
    else:
        r = requests.get('http://127.0.0.1:10000/dynamic?upstream=backends&server=' +
application + '&add=')
    return r.text


@app.route('/api/backends/<application>/add')
def add(application):
    r = requests.get('http://127.0.0.1:10000/dynamic?upstream=backends&server=' +
application + '&add=')
    return r.text


@app.route('/api/backends/<application>/remove')
def remove(application):
    appname = request.args.get('appname')
    version = request.args.get('version')
    terminate_container(appname, version)
    r = requests.get('http://127.0.0.1:10000/dynamic?upstream=backends&server=' +
application + '&remove=')
    terminate_image(appname, version)
    release_app_port(appname, version, application.split(':')[1])
    return r.text


@app.route('/api/backends/stop')
def stop():
```

```
    appname = request.args.get('appname')
    version = request.args.get('version')
    terminate_container(appname, version)
    return "container stopped and removed"


@app.route('/api/backends/start')
def start():
    appname = request.args.get('appname')
    version = request.args.get('version')
    if not is_running(appname, version):
        port = get_port_for_stopped_app(appname, version)['port']
        run_container(appname, version, port)
    return "container started"


@app.route('/api/test')
def test():
    var = is_running("myapp", "1.0.0")
    return "test"


@app.route('/api/deploy', methods=['POST'])
def deploy_file():
    if request.method == 'POST':
        file = request.files['file']
        if not request.args.get('version'):
            return "version is empty"
        if not request.args.get('name'):
            return "app name is empty"
        version = request.args.get('version')
        appname = request.args.get('name')
        create_workspace(appname, version)
        copy_assets(appname, version)
        filename = secure_filename(file.filename)
        # saving file to workspace
        file.save(os.path.join('workspace/' + appname + '/' + version, filename))
        # create docker file
        create_docker_file(appname, version)
        # get an available port
        port = get_available_port()
        # reserve port
        reserve_app_port(appname, version, port)
        # build image
```

```
        build_docker_image(appname, version)
        # clean workspace
        clean_assets(appname, version)
        run_container(appname, version, port)
        add_to_proxy(port)
        return 'file deployed successfully'


def add_to_proxy(port):
    requests.get('http://127.0.0.1:10000/dynamic?upstream=backends&server=127.0.0.1:'      +
str(port) + '&add=')
    requests.get('http://127.0.0.1:10000/dynamic?upstream=backends&server=127.0.0.1:'      +
str(port) + '&down=')


def run_container(appname, version, port):
    client = docker.from_env()
    client.containers.run(appname + ':' + version, 'tomcat/bin/catalina.sh run', detach=True,
                          ports={'8080/tcp': port})


def copy_assets(appname, version):
    copy_tree('assets/jre', 'workspace/' + appname + '/' + version + '/jre')
    copy_tree('assets/tomcat', 'workspace/' + appname + '/' + version + '/tomcat')


def clean_assets(appname, version):
    remove_tree('workspace/' + appname + '/' + version + '/tomcat')
    remove_tree('workspace/' + appname + '/' + version + '/jre')


def build_docker_image(appname, version):
    client = docker.from_env()
    client.images.build(path='workspace/' + appname + '/' + version,
                        tag=appname + ':' + version,
                        rm=True)


def terminate_container(appname, version):
    client = docker.from_env()
    containers = client.containers.list(all=True)
    i = 0
    while i < len(containers):
        tag = containers[i].attrs['Config']['Image']
```

```
        tag_parts = tag.split(":")
        if appname == tag_parts[0] and version == tag_parts[1]:
            # stop container
            containers[i].kill()
            containers[i].remove()
            print("terminated " + appname + ":" + version)
            break
        i = i + 1


def terminate_image(appname, version):
    client = docker.from_env()
    images = client.images.list(all=True)
    i = 0
    while i < len(images):
        if len(images[i].tags) > 0:
            for tag in images[i].tags:
                tag_parts = tag.split(":")
                if appname == tag_parts[0] and version == tag_parts[1]:
                    # remove image
                    client.images.remove(image=appname + ':' + version, force=True)
                    print("terminated image " + appname + ":" + version)
                    break
        i = i + 1


def create_docker_file(appname, version):
    # port = get_available_port()
    f = open('workspace/' + appname + '/' + version + '/Dockerfile', 'w')
    f.write('# Tomcat 8 customized\n')
    f.write('#')
    f.write('# VERSION               0.0.1\n')
    f.write('\n')
    f.write('FROM      ubuntu\n')
    f.write('LABEL Description="This image is used to run a customized tomcat server" Version="1.0"\n')
    f.write('ADD tomcat /tomcat\n')
    f.write('ADD jre /jre\n')
    f.write('ADD *.war /tomcat/webapps/\n')
    f.write('ENV JAVA_HOME /jre\n')
    f.write('#ENV JAVA_OPTS -Dport.shutdown=8065 -Dport.http=8060\n')
#           f.write('#RUN sed "s/8080/' + str(port) + '/g" < /tomcat/conf/server.xml > /tmp/server.xml\n')
#    f.write('#RUN cp /tmp/server.xml /tomcat/conf/server.xml\n')
```

```
    f.write('EXPOSE 8080')
    f.close()


def reserve_app_port(appname, version, port):
    f = open('conf/app_ports', 'a')
    f.write(str(appname) + ',' + str(version) + ',' + str(port) + '\n')
    f.close()


def release_app_port(appname, version, port):
    content_to_write = ''
    with open('conf/app_ports') as f:
        content = f.read().splitlines()
    i = 0
    while i < len(content):
        parts = content[i].split(',')
        if not (parts[0] == appname and parts[1] == version and parts[2] == str(port)):
            content_to_write += content[i] + '\n'
        i = i + 1
    f = open('conf/app_ports', 'w')
    f.write(content_to_write)
    f.close()


def is_port_taken(port):
    with open('conf/app_ports') as f:
        content = f.read().splitlines()
    i = 0
    while i < len(content):
        parts = content[i].split(',')
        if parts[2] == str(port):
            return True
        i = i + 1
    return False


def get_app_for_port(port):
    with open('conf/app_ports') as f:
        content = f.read().splitlines()
    i = 0
    while i < len(content):
        parts = content[i].split(',')
        if parts[2] == str(port):
```

```
            return {'appname': parts[0], 'version': parts[1], 'port': port}
        i = i + 1
    print("couldn't find app for the port")
    return {'appname': '', 'version': '', 'port': port}


def get_port_for_stopped_app(appname, version):
    with open('conf/app_ports') as f:
        content = f.read().splitlines()
    i = 0
    while i < len(content):
        parts = content[i].split(',')
        if parts[0] == appname and parts[1] == version:
            return {'appname': parts[0], 'version': parts[1], 'port': parts[2]}
        i = i + 1
    print("couldn't find app for the port")
    return {'appname': '', 'version': '', 'port': 0}


def get_available_port():
    p = 8300
    while p < 8500:
        p = p + 1
        if not is_port_taken(p):
            return p
    print("Error! Out of ports")


#   creating workspace for the app and version
def create_workspace(appname, version):
    if not os.path.exists('workspace/' + appname):
        os.makedirs('workspace/' + appname)
    if not os.path.exists('workspace/' + appname + '/' + version):
        os.makedirs('workspace/' + appname + '/' + version)


def is_running(appname, version):
    client = docker.from_env()
    containers = client.containers.list(all=True)
    i = 0
    while i < len(containers):
        tag = containers[i].attrs['Config']['Image']
        tag_parts = tag.split(':')
        if appname == tag_parts[0] and version == tag_parts[1]:
```

```
            return True
        i = i + 1
    return False


if __name__ == '__main__':
    app.run(debug=True)
```