TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

MAXIM TUOVINEN

REDUCING DOWNTIME DURING SOFTWARE DEPLOYMENT

Master of Science thesis

Examiners: Professor Kari Systä and
Professor Markku Renfors
Examiners and topic approved in the
Faculty Council of the Faculty of
Computing and Electrical Engineering meeting 8.4.2015

ABSTRACT

MAXIM TUOVINEN: Reducing Downtime during Software Deployment
Tampere University of technology
Master of Science Thesis, 49 pages.
September 2015
Master's Degree Programme in Information Technology
Major: Communication Systems and Networks
Examiners: Prof. Kari Systä and Prof. Markku Renfors
Keywords: software downtime, automation, configuration management, virtualization, Puppet.

Software downtime refers to the time when software is unavailable or its operation is prevented for some reason. New software installations, migrations and upgrades cause downtime in the system which leads to loss in revenue and reduced general level of service for the company. Difficult part is finding the right set of tools and practices that minimize this downtime at different stages of software's deployment. Choosing the right tools that are compatible with company's infrastructure is also challenging and typically happens through trial and error.

The focus of this thesis is to study different means to reduce downtime during software deployment. Study focuses on reviewing tools and practices that can be used to deliver and deploy software to a production environment without service interruption. Research was carried out as literature review by examining the documents and materials from different sources.

First part of the thesis presents the main factors behind downtime and different automation practices. Tools studied in this thesis are configuration management tool Puppet, hypervisor based virtualization and Docker tool. Second part presents a developed software deployment framework which models software delivery process from development to production. Presented framework is composed of practices and tools that are presented in this thesis.

TIIVISTELMÄ

MAXIM TUOVINEN: Reducing Downtime during Software Deployment
Tampereen teknillinen yliopisto
Diplomityö, 49 sivua,
Syyskuu 2015
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Communication Systems and Networks
Tarkastaja: Prof. Kari Systä ja Prof. Markku Renfors
Avainsanat: ohjelmiston hukka-aika, automatisointi, konfigurointi, virtualisointi, Puppet.

Tuotantoympäristössä olevalla ohjelmistojen hukka-ajalla tarkoitetaan aikaa, jolloin kyseistä ohjelmistoa ei voida käyttää tai sen toiminta on estynyt tietystä syystä. Uuden ohjelmiston asennus, siirtäminen ja päivittäminen johtavat hukka-aikaan, mikä aiheuttaa yritykselle taloudellista häviötä ja heikentää yleistä palvelutasoa. Haastavinta on löytää oikeat työkalut ja toimintamenetelmät, jotka minimoivat hukka-ajan eri ohjelmiston käyttöönottovaiheissa. Ohjelmiston nopea käyttöönotto riippuu yrityksen prosesseista ja käytetyistä työkaluista. Jälkimmäisen optimointi ja yhteensovittaminen osana muita työkaluja eivät aina ole selkäitä ja tapahtuu tyypillisesti yrityksen ja erehdyksen kautta.

Tässä diplomityössä tutkitaan keinoja, joilla hukka-aikaa voidaan pienentää uuden ohjelmiston asennusvaiheessa. Tutkimuksen tavoitteena on tutkia työkaluja ja toimintatapoja, joilla ohjelmisto voidaan toimittaa ja ottaa käyttöön tuotantoympäristössä ilman tarpeettomia palvelukatkoja. Työ perustuu kirjallisuusselvitykseen tutkimalla eri lähteiden dokumentteja ja aineistoja.

Työssä käsitellään keskeisimmät syyt ohjelmiston hukka-ajalle ja esitetään eri automaation toimintatapoja. Työkaluista tutkittiin konfigurointityökalua Puppet, virtualisointia ja virtualisointityökalua Docker. Tutkimukseen pohjautuen esitetään toimintamalli, joka on suunniteltu ohjelmiston toimitusketjuksi kehitysympäristöstä asiakkaan toimintaympäristöön. Kehitetty malli koostuu toimintatavoista ja työkaluista, joita tässä työssä esitetään.

PREFACE

This thesis is a result of the work started in Nokia Networks at Tampere Finland. I would like to sincerely thank my supervisor at Nokia Networks Timo Vehmaro for granting me this opportunity to work in such well-known and respected organization.

I would also like to thank Professor Kari Systä at the Tampere University of Technology for his valuable comments and guidance during the writing process. Finally, I want to take the opportunity to thank my family, friends and amazing colleagues for their constant support and encouragement on this journey. Thanks for making this time a rich and wonderful experience!

Tampere, September 2015

Maxim Tuovinen

CONTENTS

1. INTRODUCTION

Deploying software systems with zero downtime is important for large companies to ensure that end users have access to the system all the time. Ideally, the service should to be running throughout the year without any interruptions. However scheduled maintenances and unexpected events cause the system to fail or to be out of order. Completely removing software downtime can be achieved through proper planning, testing, and tools. In order to achieve zero downtime or to minimize it, the system must have a proper configuration and automation backing it up. Using a right mix of development tools and practices is key when reducing downtime during installation.

This thesis was started as a research project at Nokia Networks R&D department. The general purpose of the project was to investigate means to reduce software downtime, since any kind of downtime in the organization caused major financial losses. Application area of this research are large software systems that provide critical services for the users where downtime of the service is minimized. An example of such service are telecommunication service providers. Based on this the research topics are listed below.

- Investigate available technologies and tools that can be used to reduce software downtime.
- What deployment methods businesses use to deploy their applications?

The software deployment process was inspected from an installation point of view, where a new software is installed to customer's site. Installation means in this context both clean installations, where a completely new software is installed to the customer, or upgrade, in which the current software is upgraded to a newer version. Research work was done around practices and tools that can be used during the deployment process. Study included continuous deployment practices and configuration management tool Puppet. Research also included virtualization techniques and an overview of Docker virtualization tool is given. The main research problem for this thesis is defined as:

1. *How can we deploy large software to a customer system as fast as possible and how to do it in real time so that users perceive second to none downtime in the system?*

The research was carried out as a literature survey on Puppet and Docker tools and continuous delivery practices by evaluating their feasibility for software deployment. Puppet

tool was selected because it provides a high level of abstraction in configuration management and it scales well with large IT infrastructure. High abstraction means that Puppet can manage large portions of the IT infrastructure including servers, network devices, applications and more. Docker tool is a disruptive technology in virtualization market and it was selected because it provides an alternative virtualization method with its own advantages for faster software deployment. At the end of this thesis a software deployment framework is presented which utilizes automation patterns, virtualization and Puppet tool. The actual implementation of the framework is not part of this thesis but it is evaluated based on the requirements for the framework and literature documents.

The structure of this thesis is divided into seven chapters and is organized as follows. Chapter 2 is the introduction of software downtime, different sources of downtime and explains why it should be avoided in companies. Chapter 3 starts addressing the research problem of this thesis by introducing automation practices. It explains why automation is required for deployment and goes through automation patterns. An overview of deployment pipeline is presented which acts as software deployment pattern for the remainder of this thesis. At the end of the chapter, a summary is given with patterns for automated infrastructure. Chapter 4 covers configuration management tool Puppet. Chapter 5 describes hypervisor virtualization and introduction to Linux containers. In this chapter an overview of Docker is given. Chapter 6 introduces different deployment methods companies can use to deploy software. It also introduces proposed software deployment framework that is based on the tools and practices described in this thesis. Evaluation of the framework is given. Finally, Chapter 7 sums up this thesis with conclusions and future work.

2. SOFTWARE DOWNTIME

Big enterprise software systems are very complex. They require planning, work, testing, and a general ongoing effort by knowledgeable people at every phase of implementation. Software must be tolerant to failures and inappropriate inputs as well as hardware must be able to automatically recover from failures. [1] If not carefully planned, unexpected system failures will cause the loss of revenue and degrade user experience especially in industries that rely on 24-hour service. Success of reducing software downtime depends on the right set of technologies in the organization and collaboration between responsible personnel.

Achieving complete zero downtime is possible but is a costly operation. Typically the business needs to evaluate what is the maximum available downtime that the business can tolerate without having too huge loss in business and services to its customers. Most businesses don't usually put emphasis on this as they mostly haven't experienced any catastrophic failure and put a value on it. Hence businesses usually don't allow any downtime in their production environment. During unplanned downtime, if company doesn't have any repairing policy the result on the business will be severe. The business needs to estimate the acceptable level of downtime that it can endure and build own solution on top of that assumption. Downtime can be reduced by using of proper hardware, offsite replication and virtualization. In this chapter importance of software downtime is discussed as well as different sources of downtime.

2.1 Motivation

In this thesis downtime is defined from end users point of view, where if the end user cannot access the system or get the job done on time, then the system is considered being down. Typical sources of downtime come from software or hardware failures. Factors that may be causing this are such as the failure of the server itself, disks, the network, operating system (OS), or key applications. Downtime doesn't necessarily require the failure of a node or hardware but degradation of performance can also be included. This can include slow server or network connection or data inaccessibility. [1]

Impact of software downtime on the business depends on multiple factors. Factors that affect are such as lost money transactions, idle time of production, required repairs to get the system back online and idle time of employees. Depending on the size of production environment the lost revenue can wary from thousands of dollar to a million. For example if the business uses only one server in production, then if this server happens to go down then the whole production goes down and everything is in standby. This affect is less severe on businesses that have multiple servers in production and have redundant servers

on standby. Now if one server happens to go down then its impact is less harmful as another servers will keep the production running.

The cost of downtime can be divided into two categories, direct and indirect costs. Directs costs means all that lost revenue that would be have been received if the production was operating normally. This cost depends upon the nature of the company's business, the size of the company, and the criticality of IT systems to primary revenue generating processes [29]. The amount of loss revenue in different industries is approximated in Table 2.1.

Table 2.1. *Estimation of direct costs of downtime in different industries [29].*

Industry	**Average Cost / hour**
Brokerage services	$6.48 million
Energy	$2.8 million
Credit card	$2.58 million
Telecomm	$2 million
Manufacturing	$1.6 million
Retail	$1.1 million
Health Care	$636 000
Media	$90 000

Downtime causes indirect costs as well. These effects are [1]:

- Bad customer experience. If the site is down, or the customer cannot use the service properly then he will have a bad experience with the service. Depending on the past experience he might completely stop using company's services and go away. Negative experience will generate bad word-of-mouth among customers which further damages the company's reputation. This is especially problematic if bad word-of-mouth ends up on the internet where it can spread widely and fast. Losing existing customers and bringing in new ones requires much more time and money than retaining old customers.
- Bad publicity. When large businesses experience system downtime, then chances are that it will make it to the news. Bad press will injure company's reputation and can also affect the market of the business. Typically bad publicity will affect the stock price as in results the stock price will drop. This can further effect on stock holders as they may perceive corporate weakness and sell the company's stock.
- Decreased employee morale. Bad publicity, customer complaints and stock drops effect on the morale of employees inside the company. This can result in some employees leaving the company and others may not work as hard if they believe that things are going bad. If people are leaving the enterprise, then it can further

hurt the company's reputation by reducing its attractiveness as possible employer to future workers.

2.2 Types of software downtime

2.2.1 Unplanned software downtime

Unplanned software downtime is in many cases the result of unexpected software behavior, hardware failure, human made error, power failure and so on. These outages are characterized by the fact that they happen in uncontrolled matter and are hard to predict. Different sources behind unplanned downtime are presented in Figure 2.1.

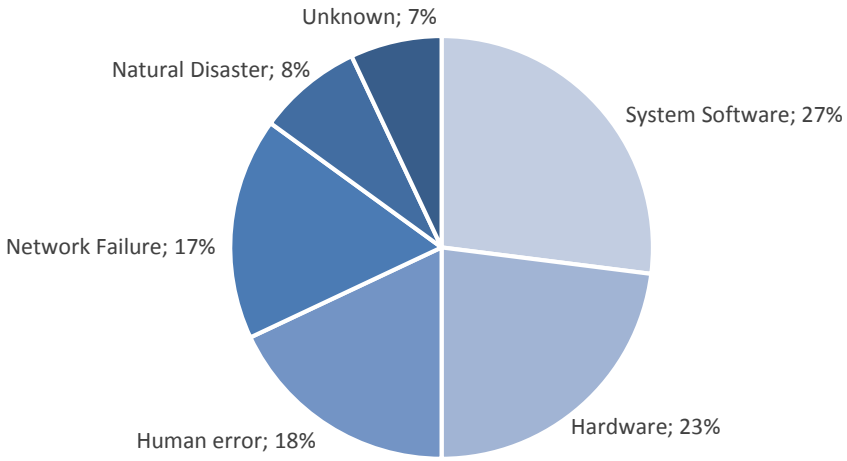

Figure 2.1. *Different causes for software's unplanned downtime. [1]*

The percent values can vary based on different organizations but the major factors stay the same. Three major factors behind unplanned downtime are system and hardware failures and human errors. Main sources of downtime [1], [2], [3] are:

- System Software**.** Software bugs are source of failures and can be difficult to debug depending on the complexity of the system. Over a long period of time the software will show degrading performance and increasing failure occurrence rate. This is due to error conditions that occur over time and eventually lead the system to failure. Sources that can cause outages can range from several layers of a complex software system, ranging from the OS to the user application level. Restarting the software from a clean state is one way to reduce this degradation.
- Hardware. Hardware failures are one the most common reasons behind failures. If hardware outage occurs, it may take a long time for a new replacement or if there isn't any redundancy built in. Components with moving parts are the most prone to hardware failure. Typically, this includes hard drive disks, fans and

power supplies that have moving parts with high speeds and complexity. To combat these shortcomings the system should have extra fans, extra power supplies and good hardware diagnostics tools to detect problems once they occur.

- Human error. Humans are prone on making manual errors during configuring or installing the product especially if there are many steps during deployment. Also errors occur as humans don't completely understand the way the system operates. Deploying a large system is in many cases manually intensive process where the chance for missing a simple configuration or configuring in wrong manner is high. In the optimum scenario there is no human interaction during deployment or to be as small as possible and letting the software configure itself though the specified rules.

2.2.2 Planned downtime

Planned downtime is often referred to as scheduled maintenance. Planned downtime happens in a controlled manner where necessary procedures are carried out to the system either to repair it or adding new functions. This means updating or replacing current hardware, components, updating software version or bringing in completely new system online. Typical reasons for planned downtime are:

- Replacing hardware part
- Firmware upgrade
- Upgrade/exchange the storage
- Configuration of the connection pool
- Configuration of the cluster
- Upgrading the cluster software
- Upgrading the database software
- Deploying a new version of software
- Moving application to another data center

Scheduled maintenances are typically performed during the time when the usage of the system is low so that the effect of downtime to the users is minimized [1]. Planned maintenances occur regularly and are scheduled weeks or months in advance. During unexpected outages, scheduled downtime is required to restore production environment to its normal state. This further increases the total downtime of the system as once unplanned downtime occurs then it has to be repaired during scheduled downtime. Reducing hardware maintenances can be done by using hardware from certified vendors with properly tested hardware.

Planned maintenances can be a source of downtime since these procedures may fail depending on their complexity. These failures will cause unexpected downtime in the service. Success of planned maintenances is influenced by the following factors [42]:

- Automated process. Easy and repetitive tasks are automated for reduced human involvement to lower the risk of human error. Manual procedures need to be clearly documented to reduce human error when executing procedures.
- Many small steps. Small simple procedures are more likely to succeed than one large step at the time. For example, a software upgrade that does not require a database schema change and data migration is more likely to succeed than an upgrade that has major database changes.
- Validated scripts. Scripts are thoroughly tested to uncover possible bottlenecks and failure scenarios that may occur during the procedures. Quality scripts have fault tolerance built inside them to enable rollback and safe stopping points in case of unexpected errors.

Software upgrade is a common procedure that is carried out during scheduled maintenance. Upgrades are needed to implement new features in the application, in order to meet changing user requirements [43]. Typical upgrade process consists of bringing the node offline, performing the upgrade and restarting the service. During upgrade the service traffic is routed to another active node so that the service is not interrupted [42]. This type of upgrade is called live upgrade in which the service is accessible during the maintenance.

2.3 Problem with manual deployments

Nowadays modern applications of any size are complex to deploy as they have many files associated and dependencies between them. Typically organizations deploy their application manually and are performed by team or set of individuals. Each step during deployment if done manually is prone to human error. Problems associated with manual releasing are presented. [2]

- When deployments aren’t fully automated, each deployment is slightly different with different errors. Debugging these errors is a time consuming task and bugs can be hard to track down.
- A manual deployment process has to be properly documented. Maintaining the documentation is a complex and time-consuming task involving collaboration between several people, so the documentation is generally incomplete or out-of-date at any given time. Automated deployment scripts serve as documentation, and it will always be up-to-date and complete, or the deployment will not work.
- Manual deployments depend on the deployment staff and their expertise.

- Performing manual deployments contains a lot of repetitive tasks that require certain degree of expertise. Asking experts to do boring and repetitive, and yet technically demanding tasks is increases the risk of human made error Automating deployments frees staff to work on other higher-value activities.
- Deploying manually is a time-consuming and expensive process to do. Also making sure that it works can only be done by performing it. An automated deployment process is cheap and easy to test.
- With a manual process, there is no guarantee that the documentation has been followed. Only an automated process is fully auditable.

Automation is used to lower software cycle time. Cycle time means all the steps that the software has to undergo from development to production environment. Since errors will occur in production, it is vital to fix these and push the application back into live environment as fast as possible. Without automation, building, testing and deploying our software will take more time and effort and increases overall costs.

2.4 Software dependencies

Software dependency means those components, packages, files and libraries that the software depends on in order to function. These dependencies need to be declared and be available during installation process so that no errors occur. Problems may occur if the software relies on a specific version of dependent software, and that dependent software is not available on user's machine [45]. Software might function properly in development environment, but deployment in the target environment can fail due to different dependency problems. Conflicting versioning, different operating systems and environments can all break the installation and cause downtime in the system.

During deployment, the software package must be properly built with required components and their dependencies available. UNIX systems such as Linux use a package manager for deploying new software packages and for maintaining the installation. Package managers define how a software package should be built and which packages it depends on. For example, RedHat-based systems use `rpm` packages and `yum` to install them, where the build specification is detailed in the RPM `spec` files. Package managers are good to keep the software up to date and declaring dependencies but have limitations to support different software builds or specific compliers as the manager takes care of specific version of the software. Other versions need own package manager. [46]

Different software components are often developed by different groups of people and their dependencies are not clearly stated [16]. This can make software deployment a risky process as the installation might not be successful and can disrupt existing system operation. Application build process uses libraries, binaries, files and source code or something

else. Once a new build is created from the source code, it can have dependencies to specific libraries and components. If versioning for these components changes then application build process can fail as the libraries it dependents on are not found. Before

Traditionally different components and libraries are separated between build-time or runtime dependencies and have them in separate directories inside a version control repository. Build-time dependencies includes all components that must be present when application is compiled. Runtime dependencies are those components that must be present when the application is running. Each file in version control should also include their version number to so that it is easier to detect which versions are up to date. Version control need to be checked from time to time so that it doesn't become too large and unclear making it difficult to know which libraries are still in use. [2]

Software dependencies are typically discovered by the analysis of source code. These relationships can be represented either by a data-related dependency or by a functional dependency. Functional dependency is where function A calls for function B and data-related dependency is where a particular data structure modified by a function and used in another function [26]. Dependency hell describes situations when an application depends upon one particular version of something, but is deployed with a different version. This is common problem where there are lots of application libraries stored in a directory without any versioning between them. Once updated versions of existing libraries are added to this repository, they would override previous libraries. This causes the application building process to fail as it depends on a certain version of libraries. Dependency types can be divided into four different categories [28]:

- Circular dependencies. This dependency problem occurs when applications component A depends on another component B, but component B also depends on component A, thus creating dependency cycle where both components depend on each other. Having circular dependency is a sign of bad design and too tight component coupling.
- Long dependencies. Application component A depends on component B, which depends on components C and so on to component X.
- Conflicting dependencies. Separate applications depend on the same component.
- Many dependencies. Application depends on multiple dependencies. This makes application downloads heavier as all components need to be downloaded.

Version control is one solution to manage dependencies. Establish a common rule to version components based on the changes made to them. Other solutions include using different directories for different builds. In this case after the code is compiled the created libraries and binaries are stored inside their own directories. Finally third party applications, such as Maven, can be used to manage and describe software's dependencies.

3. DEPLOYMENT PIPELINE FOR SOFTWARE DELIVERY

This part will begin addressing the means on how downtime can be reduced though tools and practices. This chapter will cover the use of deployment pipeline during deployment process. First an overview of the pipeline is given followed by discussion on how to implement one. End part of this chapter focuses on the patterns that can be used for reliable software deployment.

Deployment pipeline is a part of continuous delivery practice. Continuous delivery is a series of practices designed to ensure that the code can be rapidly and safely deployed to a production-like environments. New software changes are tested in the pipeline so that it is production ready and the software can be reliably deployed into production. Continuous deployment is a practice where new software features are pushed into production environment automatically. New features are tested automatically and if they pass the test then these new features get deployed to production environment. Difference between continuous delivery and deployment is the manual sign off in continuous delivery where there are many software versions in standby ready to be deployed. With continuous deployment there are no manual steps and new software is pushed to live environment automatically.

3.1 Overview

Automating software deployment gives the ability to rapidly and reliably deploy application to a selected environment. Performing deployments manually is prone to human errors as deployments consist of many steps. Ideally, the amount of manual work is minimized and automated as much as possible. Deployment pipeline is a software delivery practice which contains all the steps that the software must pass before it is production ready. The essence of deployment pipeline is to give visibility and fast feedback of software's end-to-end process thus making deployments more reliable and fast. Pipeline models all the steps that the software undergoes from development to releasing to production. Different tools are used to automate stages of the pipeline and have the ability to deploy software with the push of a button. The outcome of using deployment pipeline is an automated and reliable process for delivering software to target environment. Figure 3.1 shows an example deployment pipeline with different stages during software's lifecycle. [2]

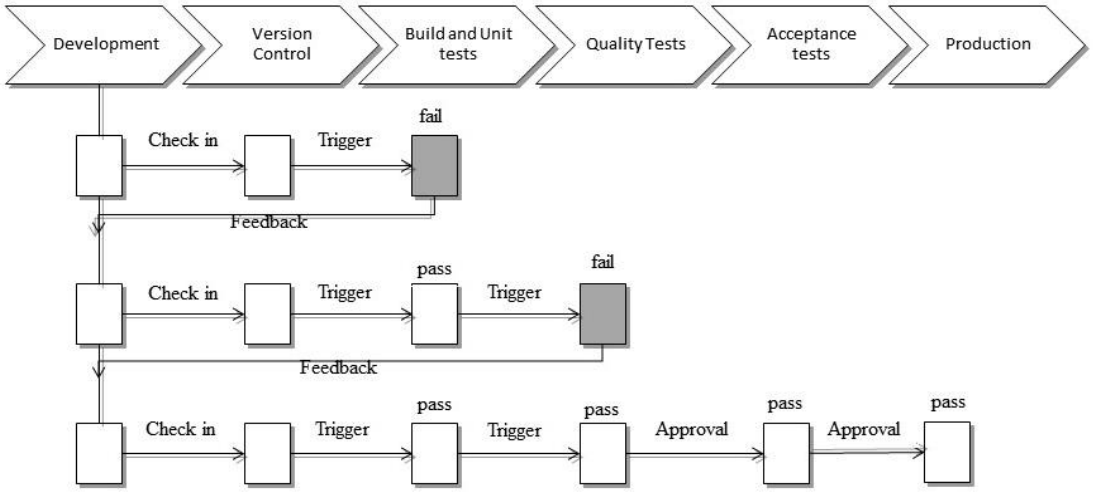

Figure 3.1. *Example deployment pipeline with different stages.*

The process starts by committing changes into version control system. This triggers the first instance, build stage, in which the code is compiled, tested and binaries created if the tests are successful. These binaries get stored then in their own repository and moved onto to the next stage. Next stage runs automated tests against the new binaries. Here the new application gets deployed into separate environments where the tests are performed. Different tests can be done simultaneously for reduced cycle time. Successful test candidates then get stored into own repository from which they are ready to be deployed to production. Feedback is sent at each stage to inform what pipeline stages the build has passed or failed. [2]

It is often easier to perform a release once the process is automated so that releases can be performed more frequently. Getting feedback that is fast and relevant is one major aspect of the pipeline. As the build passes each stage its reliability increases which means that the build become progressively more production ready. The objective of feedback is to eliminate unfit release candidates as early in the process and get feedback on the cause of failure to the teams rapidly. Any build that fails a stage in the process will not generally be promoted to the next. [2]

3.2 Implementing pipeline

Pipeline provides a framework that describes the stages that the software must undergo from source control to production. Version controlled repository is used for production ready builds which then can be released to production [9]. First step of implementation is identifying the stages that the software must go between development and releasing. These stages will serve as a base of the pipeline where each stage acts as a sign off gate for the software build. Once the build passes a set of validations it will be promoted into the next stage and so on until it release-ready. Each stage generates report that is the sent

to its own respective repository. If the build does not pass the stage it will be returned to repository. Possible stages to use in deployment pipeline are [2]:

- Unit testing. Testing certain functions of the code and verifying that they work appropriately.
- Acceptance testing. Acceptance test verifies that the new build delivers business value to the customers by showing that the requirements for the application are met.
- Performance testing. Testing the performance of the system under a particular workload and that it meets certain criteria's.
- Usability testing. Testing the software from end users point of view determining how easy the software is to use and how convenient it is.
- Code review. Searching for software bugs or any human made errors.

Above listed tests are a number of the tests that developer can decide to use for software as stages in the pipeline. Once the different stages are identified, they are arranged into an ordered pipeline in which each stage has certain inputs and outputs. Parallelization can help reducing the cycle time of the software by running different tests at the same. Different tests running in parallel is illustrated in Figure 3.2.

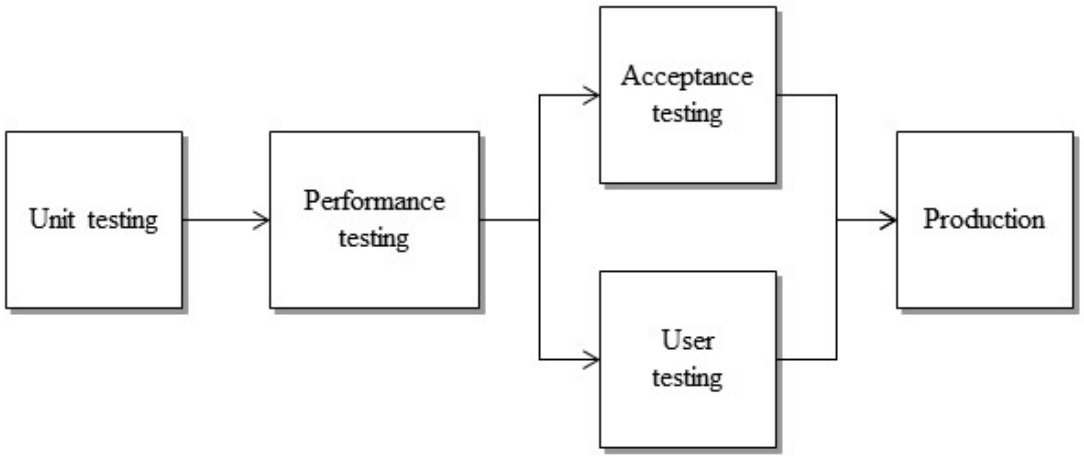

Figure 3.2. *Different tests running in parallel in deployment pipeline.*

Running acceptance and usability tests in above pipeline can speed end-to-end delivery process but can also increase total delivery time if the build has faults. For example, if both acceptance and user tests are running at the same time and if the software build happens to fail acceptance test, then the build needs to be repaired and run both acceptance and user tests again instead of verifying acceptance test followed by usability test. Pipeline presented in Figure 3.1 is very simple and shows a typical pipeline design is a balanced between tradeoffs. Table 3.1 shows common tradeoffs.

Table 3.1. *Tradeoffs when designing deployment pipeline. [9]*

Ideal	**Tradeoff**
All steps of the pipeline are automated.	Requires substantial investment in automation tools.
Human re-work is expensive.	Tests are carried out in parallel with other tests which can pose a problem as some tests might not be properly executed or some bugs are missed if the personnel must focus on many tests instead of focusing on just one. Debugging these problems later on in the pipeline will be more expensive and time consuming.
Implement lots of environments to support testing phases.	Maintaining these environments can be expensive. This is emphasized if different environments require their own dedicated hardware which in large operating environments may not be an option.

With different stages of the pipeline identified, the next step is to investigate how these stages can be fully or partially automated. Typical early stage that is automated is the software build process in which binaries are created from the source code. In this context binaries mean the created executable application from the source code which is executed on a machine. With fully automated build process, every time a commit is made in the source code the monitoring software will generate automatically a new build of the software which is stored in a version control system. Jenkins is an open source tool that triggers new build process at every commit stage in the code or at different scheduled times [10]. In the example pipeline covered in this thesis, Jenkins is used as integration server to run an automated build process of the source code. This type of automated build practice is also called continuous integration, where developers commit changes to the code which then creates new build that are saved in a shared repository [2]. Continuous integration allows early detection of problems through collaboration as developers have access to the same repository from which different builds can be tested by different developers.

Once the build process is automated the next step is to automate the deployment of application onto different machines. For example the application needs to be deployed onto dozen different machine to test it with different operating systems and third party software's. In this thesis creating different environments and deploying the application fast are carried out though virtualization and configuration management tool Puppet. This stage of the pipeline is critical as it will affect the rest stages of the pipeline significantly. This is because the operating environments created here must match the production environment so that developers can have the reliability of that if the build works on their computer then it will work on all others as well. This is ensured by using the same method to provision and deploying the application though its lifecycle to avoid the problem in

which the application works in one environment but not the other. Feasibility of using mentioned tools are addressed in Chapters 4 and 5. Once major parts of deployment and provision process are automated the application is pushed to the next stage in the pipeline where different tests are done.

Testing the application is implemented at each stage in the pipeline. Automating these tests can be done by writing scripts that perform them. Testing at each stage of the pipeline increases the reliability of the build as it fulfills the criteria's set at these stages. This can be implemented with the use of gateways at each stage. For example the integration server notices that the build is finished with certain stage and triggers build to run in another stage thus gradually moving through the whole pipeline. If the build doesn't meet the criteria's of a certain stage then the integration server will notify about it to the developer in according manner. Many businesses will likely prefer manual sign offs [9] between stages for added certainty. Ideally after each run the summary of the procedures are notified to the developer on the results of the test. Getting fast feedback of the run is vital and notification is sent immediately after each run or after the build fails the run. Motivation behind fast feedback is to identify possible problems as soon as possible and fix them.

Implementing deployment pipeline will increase the chance of delivering software reliably and fast as it gives the visibility of the whole delivery process and fast feedback. Ideally deploying a new software to production is performed by push-button deployments with little human intervention. Gateways are used in critical stages of the pipeline to ensure that the build is production ready. Each release candidate is validated that it has passed the whole pipeline successfully with possible defects optimized and fixed. Each pipeline design is different and depends on the application as there is no one-size-fits-all solution. Good pipeline consists of: proper configuration management, automated scripts for building and deploying the application and tests to prove that the application meets the requirements [2]. To summarize functionality of deployment pipeline:

- Selected stages that the software must go through to be release-ready.
- Reporting after each stage it is successful or failed or once the build fails specific stage. Each stage has certain criteria's that the software must pass.
- Visibility of the software life cycle, showing progression of builds and what changes are associated.
- Automated push-button like deployments.

3.3 Monitoring

During software deployment constant monitoring is necessary to detect defects in production and during deployment process. Different metrics are gathered and compared against baseline metrics to monitor the behavior of the new release. If something goes

wrong, the operations team is informed immediately about the incident, and use the tools to track down the root cause of the incident and fix it [2]. User traffic is also monitored to ensure that no data corruption happens when the users are switched between different production environments. Monitoring needs to be implemented to minimize the effect of unplanned downtime presented in Chapter 2. Alarms need to be triggered immediately if the service is unavailable or reduced quality of service in the production e.g. high latency in the network. Availability of the service over a certain period of time should also be monitored to estimate the complete uptime of the service during throughout the year and improve on change.

The essence of deployment pipeline is to provide feedback about the status of the software build as it passes the stages of pipeline. Monitoring different components gives the assurance that the systems performance doesn't suffer when deploying new software. Feedback is used to provide metrics, and log files about the application in production. Reports are used in development to fix bugs that were detected in a running system. [22] Following attributes can be monitored during software deployment to ensure stability of the release: [22]

- Hardware. Monitoring physical hardware parameters will reduce the number of hardware failures. Almost all modern server hardware implements the Intelligent Platform Management Interface (IPMI), which monitors voltages, temperatures, system fan speeds, peripheral health, and so forth. IPMI has four main features: monitoring the hardware, ability to restart and recover the server, logging states of the hardware and list of all monitored hardware components [30]. If any of the hardware components reaches critical level then deployment process is aborted and that component replaced before it fails.
- Middleware. Middleware means the layer of abstraction between hardware and applications. Typically these are operating systems, hypervisors and kernels. All these systems provide interfaces to get performance information such as memory usage, swap usage, disk space, I/O bandwidth, CPU usage, and so forth.
- Software. Applications should monitor things that both operations and business users care about, such as the number of business transactions, their value, conversion rate, and so forth. They should record the status of connections to external systems that they rely on.

Monitoring can be implemented through commercial and open source tools that will gather everything described above across the whole data center, store it, produce reports, graphs, and dashboards, and provide notification mechanisms. Well-known commercial monitoring tools are Microsoft's Cluster Server, HP Serviceguard and VERITAS Cluster Server. Figure 3.3 illustrates monitoring dashboard with open source tool OpenNMS [23]. When selecting a system monitoring protocol or tool, make sure to select technologies

that are mature and that have reference sites and web sites as sources of good information about successful deployments [1].

Availability Over the Past 24 Hours		
Categories	**Outages**	**Availability**
Network Interfaces	10 of 117	91.795%
Web Servers	5 of 50	91.703%
Email Servers	1 of 19	96.491%
DNS and DHCP Servers	0 of 10	92.004%
Database Servers	0 of 1	100.000%
JMX Servers	0 of 0	100.000%
Software Update	3 of 18	83.081%
Other Servers	8 of 66	90.551%
Total	**Outages**	**Availability**
Overall Service Availability	29 of 287	91.088%

Figure 3.3. *Open NMS Dashboard for monitoring application availability in production.*

3.4 Making deployments more reliable

Same deployment process should be used to deploy software to every environment, whether its development, testing or production environment. Teams need to be sure that the build and deployment process is tested effectively. Deployment process needs to be practiced multiple times in different environments so that deployment process and scripts are error-free. This makes debugging easier as when a deployment doesn't work in a particular environment the problem most likely lies in the following causes:

- A setting in application's configuration file is wrong.
- A problem with infrastructure or one of the services on which the application depends is unavailable.
- Environment has different configuration.

If the production environment is simple or the enterprise have a sufficiently large budget, there can be two exact copies of production environment where manual and automated tests are performed. This environment needs to be exact copy of the real production environment with the same configurations so that teams are confident that the changes will work in live environment. Binaries that get deployed into production should be exactly the same as those that went through the pipeline process. Recreating binaries can cause risk that some change will be introduced between the creation of the binaries and their release, resulting that the binaries released will be different from the tested ones. [2]
The following configurations should be the same [2], [6].

- Infrastructure. Make sure that network topology, firewall configurations are the same and software dependencies are clearly specified.
- Middleware. Operating system configuration, including installed patches and services are the same.
- Applications. Third party software applications and application stack is the same.

Deploying software reliably depends on careful planning between teams that are responsible for releasing. Result of good planning consists of proper documentation, automated scripts, configuration and orchestration of the whole deployment process. Complexity of the deployment depends on the size of application and its dependencies. Deployment pipeline should be used to model the test and release process for the software. As the build goes through the pipeline it must pass certain requirements that are set. When planning deployment process the following should be considered [2]:

- Configuration management strategy. Teams need to know how configurations are applied to the operating environment and how they are managed.
- Description of the technology used for deployment. What tools are used when deploying and documentation of each tool for debugging.
- Implementation of deployment pipeline. Whole deployment process is modelled as a pipeline where each step is visible. This makes deployment more reliable as possible bottlenecks can be spotted early and fixed
- Acceptance, capacity, integration, and user acceptance testing. These are necessary tests to make sure the software operates in planned matter. Teams need know how these are performed as part of the deployment.
- Rollback plan. Description of the steps required if something should go wrong during deployment and restoring the system to previous state.
- Data migrations and keeping data synchronized between environments. If application utilizes database servers then synchronization between environments must be addressed. Worst case would be users experiencing data loss or data corruption especially in money transactions.
- Problems with previous deployments and their solutions. Deploying multiple times will make deployments more reliable process as teams get more familiar with it. Each deployment should be documented so that time is minimized during debugging if that problem has been previously solved.

4. CONFIGURATION MANAGEMENT WITH PUPPET

This chapter focuses on configuration management tool Puppet to automate different deployment pipeline stages discussed in previous chapter. There are many configuration tools to choose from besides Puppet, such as Chef, Ansible and Salt. Choosing the right tool depends on the programming language and the complexity of the tool. Part of the thesis was studying on how Puppet configures new environments, installs software packages and enables necessary services. For this study Puppet was evaluated based on the literature and case studies available. Puppet was chosen because it is a mature technology and is widely used in large enterprises. Puppet supports rapid scalability in infrastructure and has flexibility in managing resources between managed devices. Puppet integrates well with other products such as VMware, which is used at Nokia Networks and with the ease of use and enhanced reporting Puppet felt as the right approach.

Puppet Lab's website states:
"Puppet is a configuration management system that allows you to define the state of your IT infrastructure, then automatically enforces the correct state. Whether you're managing just a few servers or thousands of physical and virtual machines, Puppet automates tasks that sysadmins often do manually, freeing up time and mental space so sysadmins can work on the projects that deliver greater business value." [11]

Configuration management tools work by specifying a system's setup through a set of rules. For instance, developer would specify rules for configuring the system's basic infrastructure, storage and networking, necessary applications or services installed, and the roles of the system's users. Puppet is a tool that can manage all this in an automated fashion. Puppet enables push-button deployments by executing scripts that install software packages and configure necessary services in specific environment. This reduces deployment time as manual configurations are not performed or are kept to a minimum. From fault tolerance perspective Puppet can be used to provision new environments. For example if a machine goes down, then Puppet is used to automatically redeploy the application to a new machine with exact configurations as previous one. These features support the idea for zero downtime deployments. Puppet allows automating the following procedures.

- Installing software
- Configuring the software
- Managing services running on the machine
- Setting up users on the machine

4.1 Overview

Puppet is a Ruby based program that is developed by Puppet labs. Puppet can be used to manage configuration on UNIX, Linux and Microsoft Windows platforms. Puppet is often used to manage a node throughout its lifecycle: from initial installation, to upgrades, maintenance, migrations, and to software's end-of-life. [8] Puppet provides a web-interface that allows to manage state of monitored nodes with a graphical interface. On the contrast in most commercial tools all management is done through command line and graphical interfaces [38]. The language used in Puppet is declarative where the user declares the state of the machine. Puppet uses a layer of abstraction that will take care on the necessary actions needed to achieve that state.

Puppet architecture works in client-server fashion. Central server runs Puppet master daemon which has a list of all machines that it controls. Each of the controlled machines runs Puppet agent client that communicates with the master node. Puppet master contains all configurations in modules that are then distributed to agent nodes by pull operation. The agents talk to the Puppet master over client-certificate authenticated SSL and the master hands out requested configuration catalog. In its default configuration, the agents work in a polling mode and check in for catalog updates every 30 minutes [7]. Agents communicate with the server to ensure that the environment that is under Puppet's control is synchronized with the latest version of the configuration. Puppet notifies the deployment agents that a new specification is available and ready to be pulled. Workflow between Puppet agent and master is illustrated in Figure 4.1.

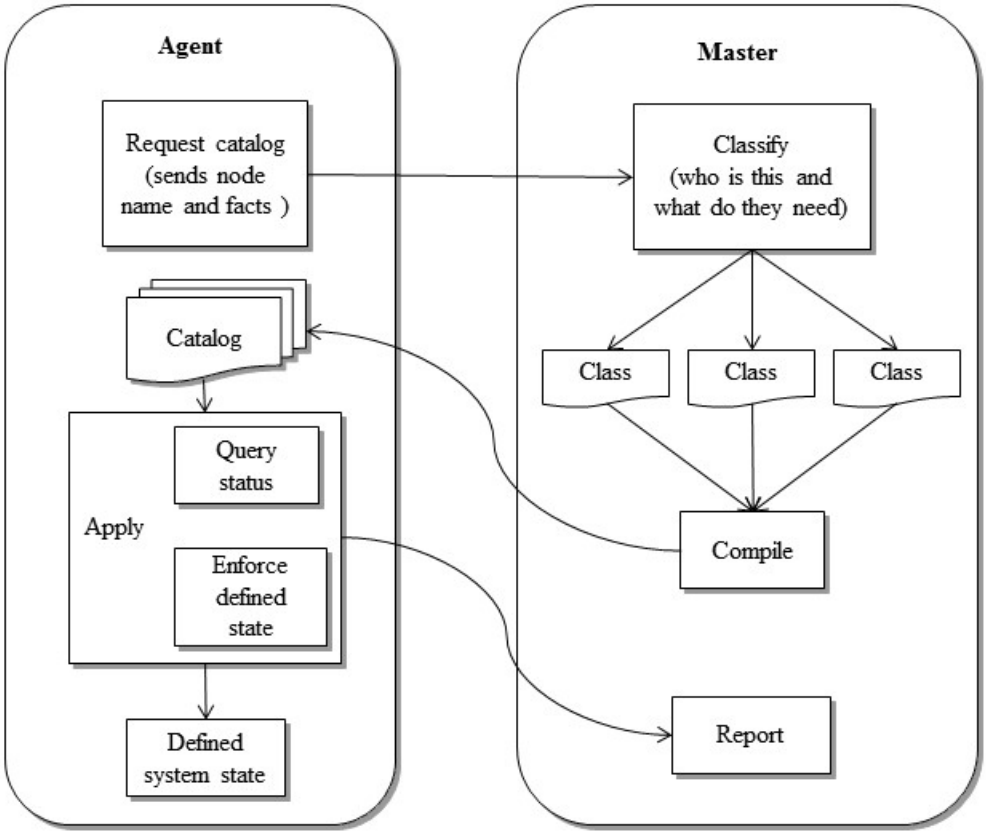

Figure 4.1. *Workflow between Puppet agent and master server.*

Puppet agent first contacts the master server and requests for configurations. Puppet master takes this information and compiles it into a catalog that is specific for the operating system it is being applied to. This catalog is then applied to machine by the agent client. [20] At the end of the configuration Puppet will generate a report on the changes made. When a configuration changes, the master client will propagate new changes to all the clients that need to be updated. Client agent installs and configures the applications, and restarts the service if necessary. Configuration language used is declarative, and describes the desired end state of the machine. This allows configuring from any starting state, either a fresh copy of or existing machine. [2]

4.2 Puppet manifests

Puppet configuration files are called manifests which are grouped inside modules. These manifests are written by domain-specific language to describe machine configurations. [20] Modules hold all the resources related to specific configuration, this includes manifests, templates and other files necessary. Modules can be used to configure new environment from their vanilla state. Listing 4.1 shows a general form that Puppet uses to declare resources.

```
type {'title':
     attribute => value,
}
```

Listing 4.1. *General form of Puppet resource.*

Resource consists of three components: type, a title, and a set of attributes. Resource type describes what kind of resource is used configured. Typical resource types: files on disk, cron jobs, user accounts, services, and software packages. Title is used to identify specific resource to Puppet. Attributes are used to describe the desired state of the resource. [11] Listing 4.2 shows an example configuration manifest file that configures postfix service. Postfix is a mail transfer agent used in Linux environment to route mails. Once a postfix module is created, user will then modify the manifest file with specified rules on how to install the resource.

```
# /etc/puppet/modules/postfix/manifests/postfix.pp

class postfix {
     package { postfix:
          ensure => installed
     }
     service { postfix:
          ensure => running,
          enable => true
     }
}
```

Listing 4.2 *Puppet manifest file for installing Postfix service [2].*

First line specifies the root directory of the manifest file. The `class` keyword declares a group of resources that user wants to apply. The `package` statement checks if the postfix package is installed, if not then it is installed. The `service` statement ensures that the Postfix service is enabled and running. The `ensure` attribute controls whether or not the service should be running. Attribute `enable => true` starts the service once the machine is booted. If this is not desired the attribute is changed to `enable => false`.

Above manifest is a simple classification of how Puppet declares desired state of the machine. It describes what packages and services are need to be running. Puppet will take care itself on how the installing procedures are carried out so that above state is achieved. Next manifest file showcases flexibility on how user can declare additional rules when configuring node. Listing 4.3 describes how sudo module is configured. Sudo is a program designed to give limited root privileges to users.

```
# /etc/puppet/modules/apt/manifests/init.pp

class sudo {
     package { sudo:
          ensure => present,
     }
     if $operatingsystem == "Ubuntu" {
          package { "sudo-ldap":
               ensure => present,
               require => Package["sudo"],
          }
     }
     file { "/etc/sudoers":
          owner => "root",
          group => "root",
          mode => 0440,
          source => "puppet://$puppetserver/mod-
          ules/sudo/etc/sudoers",
          require => Package["sudo"],
     }
   }
```

Listing 4.3. *Example Puppet manifest file for configuring server to fetch application builds from remote repository.*

Above module contains a single class called sudo with three resources declared. First package resource ensures that sudo package is installed. Second resource uses Puppet's `if` syntax for installing sudo-ldap package if the operating system is Ubuntu. Puppet will check the value of the operating system fact for each connecting client. The `require` parameter creates a dependency relationship between packages. Here the sudo package is required by the sudo-ldap package. Hence, the sudo package must be installed first before installing sudo-ldap. Last resource manages /etc/sudoers file and its first three attributes specify the owner, group and permissions of the file. File is owned by the root user and group and has its mode set to 0440 which gives read permissions. Next declaration `source`, allows Puppet to retrieve a file from the Puppet file server and deliver it to the client. It contains name of the file server and the location of the file. [8]

Last example shows how Puppet declares how resources are installed to specific node. Node declarations identify a specific machine by its hostname, and tell Puppet which resources should be applied to that node. Listing 4.4 specifies Posftfix service to be installed on nodes dev.example.com. Listing 4.2 specifies the installing procedures for postfix resource.

```
# /etc/puppet/manifests/nodes.pp
node 'dev.example.com' {
     include postfix
}
```

Listing 4.4. *Manifest file that installs Postfix service to a specific node.*

By default Puppet applies resources in the order they are declared in the manifest. To avoid software dependency problems, described in chapter 2.4, resources can be managed in specific order. Puppet uses parameters to manage dependencies between one or more target resources. Target resource is the class resource, for example class `sudo` from listing 4.3. Common parameters to manage dependencies are [11]:

- `Before`. Applies a resource before the target resource.
- `Require`. Applies a resource after the target resource. For example, declaring `require mysql` in the manifest file will cause the `mysql` class applied before any class resource.
- `Ensure`. Declares what service needs be running, or stopped, package installed or uninstalled and what file needs to be available.

4.3 Feedback

One of the objectives for using deployment pipeline was receiving fast feedback at the end of each stage of the pipeline. Feedback makes deployments more reliable since developer can confirm that the deployment process has been performed correctly. This feedback can be provided with report documents, emails, checking into document repositories and so forth. Report documents make debugging more bearable as good reports will include the applications information from previous and newly applied configurations.

Each time configurations are applied with Puppet agents, it will produce a detailed report. Puppet clients can be configured to send reports at the end of every configuration run and sent to master server. These reports include all of the log messages generated during the configuration run and metrics related to what happened on that run. Reports [11], [15] contain the following information:

- How long the configuration took?
- The total number of resources being managed.
- How many resources were skipped, because of either tagging or scheduling restrictions?
- Whether or not the resource was out of sync (didn't match the manifest).
- Whether or not the resource was changed.
- The number of properties (attribute values) that were out of sync.

- The number of properties that were changed.
- The total number of changes in the transaction.

These reports are in standard YAML (YAML Ain't Markup Language) [31] format and are divided into three sections: time, resource and metrics. YAML formatted reports are well supported by other tools such as Ruby. These reports can then be used by other programs that perform certain functions based on the data.

4.4 Summary

The goal of configuration management is to automate creation of new environments, configuring application and minimize any manual errors that may occur. Puppet tool enables faster software deployments by automating development, testing and production environments. Scalability ensures that Puppet can keep thousands of nodes up to date with right configurations which manually would be impossible. During unplanned downtime creating new environments with necessary services fast is key to minimize the effect of downtime by getting production up and running fast. Deploying new software with Puppet ensures that the software is installed with predefined configurations and dependencies. This will reduce the risk of human made errors during deployment which may cause downtime.

Puppet tool gives an easy method to start manage infrastructure though version control and automated workflows. Configurations are shared and accessed through directories by other developers. To get started with configuration management developers need to think small. For example looking up for any small actions which are performed manually when setting up new environment. This can be something like configuring firewall, installing users, downloading software and so on. Ideally with Puppet developers automate steps that take the most time when deploying software. Another step is to think which stages of the deployment process are most error-prone or requires manual procedures and start automating from there. This might not seem much but when performing these tasks multiple times per day on many hosts the total process will take a lot of time. Instead write a script that will perform this procedure. Automating software deployment needs to be practiced multiple times. Measure each deployment with valuable metrics such as elapsed time and improve the process from there.

5. VIRTUALIZATION AND LINUX CONTAINERS

This chapter will introduce hypervisor based virtualization and Linux container (LXC) technology. Redundancy is also discussed and shown how it can help preventing system downtime. Virtual environments are heavily used in Nokia's infrastructure, thus it was chosen as part of this thesis. Research was focused on investigating alternative methods to traditional virtualization techniques. Because of this the research was conducted around LXCs and Docker tool. VMware's virtualization was selected because it is a leader in the virtualization market [34], it supports wide variety of operating systems, hardware and has solutions regarding reduced downtime.

LXCs offer a lightweight virtualization at operating system level and runs applications inside containers which can be ported to any system that supports LXC. Virtualization gives many advantages for software deployment by creating custom environments in which application and its components are installed. First an overview of virtualization and LXCs is given. This is followed up with introduction to redundancy and how it can increase availability of the system. Docker tool is presented at the end of this chapter.

5.1 Overview

Virtualization describes the ability to run an entire virtual machine, including its own operating system, on another operating system. Virtualization uses a layer of abstraction on top of hardware resources. It enables creation of entire computer systems, called virtual machines (VM) that run multiple instances of operating systems simultaneously on a single physical machine. Each virtual machine is logically distinct, has been allocated with necessary hardware resources from the host, has its own logical assignments for the storage devices where operating system is installed, and can run installed applications within its own operating environment. [28] VMs are isolated from each other completely and can use different operating systems.

VMs are created and managed through hypervisors. Hypervisor is a layer of abstraction that allocates specific amount of hardware resources to each VM including its memory, processing power, storage space and network layer. Hypervisor has full control of the physical machine's hardware resources. [2] Figure 5.1 illustrates difference between virtualized and traditional machine architecture.

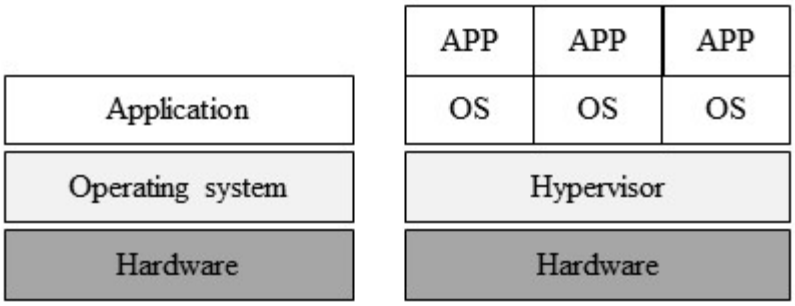

Figure 5.1. *Traditional machine architecture on the left and virtualized architecture on the right.*

The main advantages [2], [28] for using virtualized infrastructure are:

- Reduced costs. Developers may need to work with specific versions of tools, an operating system kernel, or a specific operating system distribution. Virtualization allows creation of own independent machine clusters and use them as development platforms. Reducing the amount of hardware used will reduce overall costs which overtime will result in significant cost savings.
- Improved downtime recovery. Virtualization has the ability to migrate virtual machines from one host to another during outage. Booting up a snapshot is faster than typical server thus reducing downtime. These snapshots can be taken throughout the day ensuring that data is up-to-date.
- Testing. Developers can create base images that contain proven configuration of the system. Then apply own configurations and test them. If something goes wrong or just unsatisfied with the changes then changes are erased by reverting back to the base image. Virtualization also enables speedy testing as multiple tests can run in parallel between VMs. For example, developer wants to test how the application works with different browsers, he can create own VM for each browser and run the tests at the same time instead of testing them one by one.

5.2 Virtualization vs Containers

Linux container technology is a lightweight form of virtualization that performs the same functions as a typical VM. LXC creates an environment that is close to standard VM but without the use of separate kernel and guest OS. Traditional virtualization abstracts the hardware layer of a machine, meaning that there is no need for actual physical hardware for a VM. Hypervisor allocates hardware resources to each VM and serves as an abstraction between hardware and VMs. Containers abstract the operating system layer, eliminating the need for having an OS to run application. Removing the need for additional OS will increase the performance of the system significantly.

Containers inside the host share the host's resources with all other containers and processes allowing to dynamically increase processing power if needed. VMs have limited amount of resources to use and can only use allocated amount of resources to operate. This translates into greater resource usage, redundant running processes, and performance overhead [13]. Figure 5.2 illustrates difference between LXC and VM architecture.

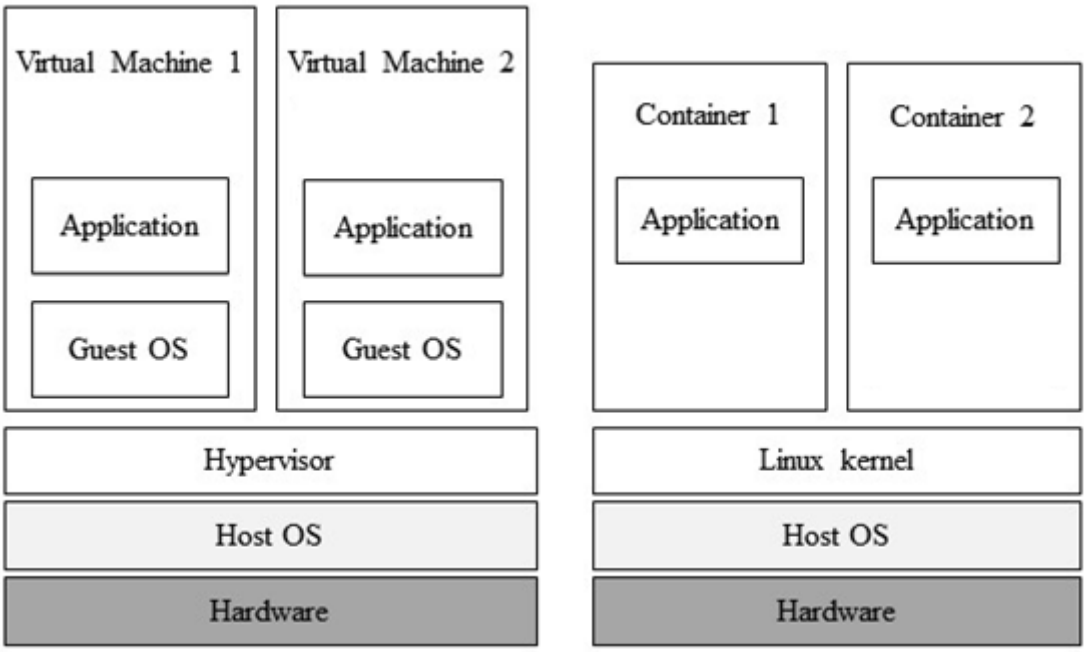

Figure 5.2. *Comparing Linux container and traditional virtual machine architecture.*

The problem with traditional VM approach is over allocation and uneven use of hardware resources. For example, if a VM uses 2 gigabytes of memory to operate but it has been allocated with 4 gigabyte of memory for its use and once the VM is booted this memory is instantly reserved. Another example is where a server uses ten VMs, each with 4 gigabytes of memory allocated to them, and once the VMs are booted a total of 40 gigabytes of memory is reserved from the hardware pool for the VMs even though each VM uses only 2 gigabytes of memory. Same principle applies for CPUs, memory and storage in VMs.

Another problem with VMs is boot ups. Since VM is treated as a normal machine it has to be rebooted after. For example a virtual server is running inside a VM and it needs to be rebooted, this boot process takes a lot of time and depending on the size of environment this downtime may not be acceptable. The extra overhead caused by guest OS and kernel is main reason for long boot ups. Speed is one main reasons why we are interested in containers as starting and rebooting applications within containers can be done much faster as there is no additional OS to reboot. Changing hardware resources for VM is by done by first bringing down the machine, reconfiguring it and rebooting. During this

planned downtime the machine may be rebooted multiple times and this sequence takes more time than LXCs.

Containers have a clear advantage over VMs because of performance improvements and reduced startup times [14]. There is nothing stopping us from having containers running inside virtual machines. Combining both techniques brings best of both worlds, VM isolation and performance of LXCs. Both techniques can be used in deployment pipeline to deploy application more rapidly to specific environment. For instance create a dedicated testing environment inside a VM that consists of containers. Different testing environments are inside own containers where tests are executed in parallel. Table 5.1 compares LXCs and virtualization from different points.

Table 5.1. *Comparison between hypervisor based virtualization and Linux containers [14].*

	Virtual Machine	**Linux Container**
Guest OS	Each VM runs on virtual hardware and Kernel is loaded into in its own memory region	All the guests share same OS and Kernel. Kernel image is loaded into the physical memory
Communication	Will be through Ethernet Devices	Standard IPC mechanisms like Signals, pipes, sockets etc.
Security	Kernel is responsible for isolating virtual environments.	Containers use namespaces to isolate processes and SELinux feature for security.
Performance	Virtual Machines suffer from a small overhead as the Machine instructions are translated from Guest OS to Host OS.	Containers provide near native performance as compared to the underlying Host OS.
Isolation	Sharing libraries, files etc between guests and between guests hosts not possible.	Subdirectories can be transparently mounted and can be shared.
Startup time	VMs take a few minutes to boot up	Containers can be booted up in a few seconds as compared to VMs.
Storage	VMs take much more storage as the whole OS kernel and its associated programs have to be installed and run	Containers take lower amount of storage as the base OS is shared

Key difference is that while the hypervisor abstracts an entire device, containers just abstract the operating system kernel. [31] LXCs eliminate the need for having OS in order to execute application. This gives immense performance boost as containers don’t have

performance overhead caused by running additional OS. VMs offer improved isolation and security than containers at the cost of performance. Inside a container the application has its own file system, storage, CPU, RAM, and so on. Only thing developer need to worry is that the application is compatible with the host OS and kernel that runs containers.

5.3 Redundancy

Virtualization provides an easy and cost effective to include redundancy in the system. Redundancy is achieved by using additional standby components in the system. If one of these components fails, then other one will take its place thus ensuring uninterrupted service. Component failures in this context can mean data corruption, connection loss, crashes, power outages, device errors. For this thesis redundancy is inspected by site-level solution where a standby VM takes over the role of a failed node. This method is powerless if the whole data center is wiped out or is fatally damaged. To cope with disaster recovery scenarios some sort of data redundancy needs to be implemented at secondary sites [49].

Using redundant VMs on separate physical hardware provides better protection against physical hardware failures. The main focus for using virtualization is that it can save a lot of money as virtualized standby nodes can be created on existing hardware instead of dedicated hardware for specific application. If one virtual system fails, then another virtual system running the same application takes over. In case of a physical hardware failure hypervisors and LXCs operation need to be moved to a failover standby node. From downtime perspective, LXCs would perform faster as there is no additional guest OS that needs to be booted [50].

Designing redundancy starts by analyzing existing system, operating environment and their usage. Typical approach includes identifying the single points of failure and adding redundancy to protect against or recover from failures. [39] Single point of failure means a critical system component in the system that in case upon failure will stop the system of operational functions [41]. Also possible bottlenecks that affect the systems performance system are considered single points of failure. During software deployment, possible failure sources can include network equipment that is responsible for switching users, shared resources be-tween multiple servers and clusters and dependencies between components.

Newer systems eliminate maintenance downtime by using load balancing and failover techniques. Two main redundancy models are used, passive redundancy and active redundancy. Passive redundancy is used to achieve high availability by including extra hardware that mitigates failure scenarios by using a backup component. Active redundancy is used to achieve high availability with no performance decline. Multiple items of the same kind incorporate into a design that includes a method for the automatic detection

of a failure and failover capabilities. Redundant components will increase the availability of services but more components means that the number of points where failures can occur increases and the system becomes more complex. [40] The system will become more homogeneous with specific type of architecture. For example the system cannot use mixed architectures between Intel and AMD processors due to hardware requirements. Different redundancy models [47], [48] are:

- Active/Standby redundancy model: In this model, one Node is active and one Node which is in standby. If active node fails then all service is failed over to the other controller. This is the most expensive method as every active node has a standby node in the background, thus requiring more hardware resources.
- Active/Active redundancy model: This model has only active nodes and no standbys. Both nodes are running critical applications at the same time. Each node acts as the standby for its partner in the cluster, while still delivering its own critical services. When one server, A, fails, its partner, B, takes over for it and begins to deliver both sets of critical services until A can be repaired and returned to service.
- N+M, Active/Standby redundancy model: It has N active nodes and M standby nodes. The number of active/standby Nodes is not limited. This model is commonly used to manage power supply outages by having backup power supplies when one supply fails [39].
- N+1, Active/Standby redundancy model: It has N active nodes and 1 standby node. If an active controller fails, all traffic served by the active node would failover to the standby node. This model requires that the traffic capacity of the standby controller is able to support the total amount of traffic distributed across all active nodes in the cluster.

5.4 Reducing unplanned and planned downtime

Unplanned downtime is reduced by using a load balancer and redundant nodes. This will help with both software and hardware failures as if the service performance starts to degrade then the standby node will take over. Load balancer is used for distributing the load across VMs so that no VM will fail because of excessive amount of requests that exhaust its resources. Production environment is formed by using VMware's virtualization where multiple VMs form a cluster. Clustering requires, at the very least, servers installed with processors of the same architecture. Servers should have identical configuration. This ensures that when a VM is migrated to a new host, minimal interruption occurs from conflicting configurations. This cluster is managed by VMware's load balancer which distributes the incoming load across virtual hosts evenly. Load balancer monitors CPU usage, memory, disks, and network and distributes the load to each VM based on these parameters. Distributing the load evenly also means that no single machine gets overloaded which can lead to a high strain of the hardware resources. Stressing the hardware

will increase the higher risk of component failure and increases the temperature which in result will require better cooling for the system. [49]

Active/standby model is selected for redundancy, where redundant VM is used to create a backup copy of hard drives and the primary VM. Only the primary VM responds to service requests, while the redundant VM is kept in sync with the primary VM in the background. If the primary fails, the secondary VM takes over immediately with no interruption in service. Since the secondary VM now becomes the primary, a new redundant VM will be created immediately and kept in sync with the new primary VM. Active/standby model provides the best reliability as the switchover between VMs can be done quickly as the data is synchronized between machines. Active/standby model is more expensive due to the fact that the standby node must be exact copy of the primary node and use a sophisticated synchronization algorithm. This model was selected because of the critical uptime requirement for the service by adding reliability at the increased cost.

Planned downtime is addressed by using live migration functionality which is built in VMware's live migration solution. Live migration means moving the application between physical machines with little to no perceived downtime in service. After successful migration, the service is running on another host and the previous host is brought offline. Once the machine is brought offline it can be modified or upgraded as explained in Chapter 2.2.2. The administrator should take down the VMs so that interruption of service is minimized.

When live migration starts, the entire state of a virtual machine is encapsulated by a set of files stored inside a shared storage between hosts. Then it makes iterative copies of a running VM's memory from its current node to another node. The purpose is to preserve the changes in the memory of the VM, each iteration copying only changes made since the last. Thus each copy is smaller than the previous one. When the size of the memory gets small enough, live migration pauses the VM and copies the last portion of changed memory. It then resumes the VM on the node that is the migration target and applies all changes in memory. The result is that the VM now runs on another node, with its memory in the state it was in when it was paused. Since the VM's folder is stored in shared storage, it does not have to be copied to the second node. Downtime occurs after the VM is paused on the original node and before it was resumed on the second node. This entire process takes less than two seconds on a Gigabit Ethernet network. Network connections and virtual identity are managed in the process. Once the new host is activated, the migration service pings the network router to ensure that it knows about the new physical location. [49], [51]

VMware's resource manager evaluates the load on the hosts and decides whether migration is necessary to meet the service requirements. VMware uses three settings for this.

First setting only warns the administrator that load balancing should take place but does not initiate balancing. Second setting moves VM to a recommended host in the cluster when VM is powered on. Third setting monitors the resource utilization and if the host is over utilized, the manager automatically moves VMs from that host to another physical host where more resources are available. Automated migration can be done either by moving VM the moment resource utilization goes above the threshold, or by waiting until overutilization has exceeded threshold substantially. An administrator can also create and apply own rules. [49], [52]

5.5 Docker Containers

Docker is a container virtualization technology that encapsulates application and all of its dependencies into a system and language-independent package. Docker creates a virtual container, inside which the application is used. This Docker container holds all necessary services and other dependencies that is needed for the application to run. [19] This container is generated into a Docker image which then can be ported to a different physical Docker environments where the application can be booted fast. Docker was selected because of its speed during application deployments. Docker has less performance overhead than traditional VMs as it doesn't use additional OS and has better hardware utilization. Docker containers offer lightweight virtualization with agile compute resource and offer more scalability than traditional virtualization. Docker helps with continuous integration and continuous deployment as developers can build stacks of Docker containers on their laptops that replicate production environments and run their application in that stack.

Docker uses a client-server architecture where user uses Docker client to communicate with the Docker daemon. Docker client sends actions to Docker daemon which then performs building, running, and distribution of Docker containers. Docker container consist of an operating system, user-added files, and meta-data. Docker image tells what the container holds, what process to run and a variety of other configuration data. When Docker runs a container from an image, it adds a read-write layer on top of the image in which the application runs. Both the Docker client and the daemon can run on the same system, or the Docker client can connect to a remote Docker daemon. Figure 5.3 illustrates Dockers client-server architecture. [12]

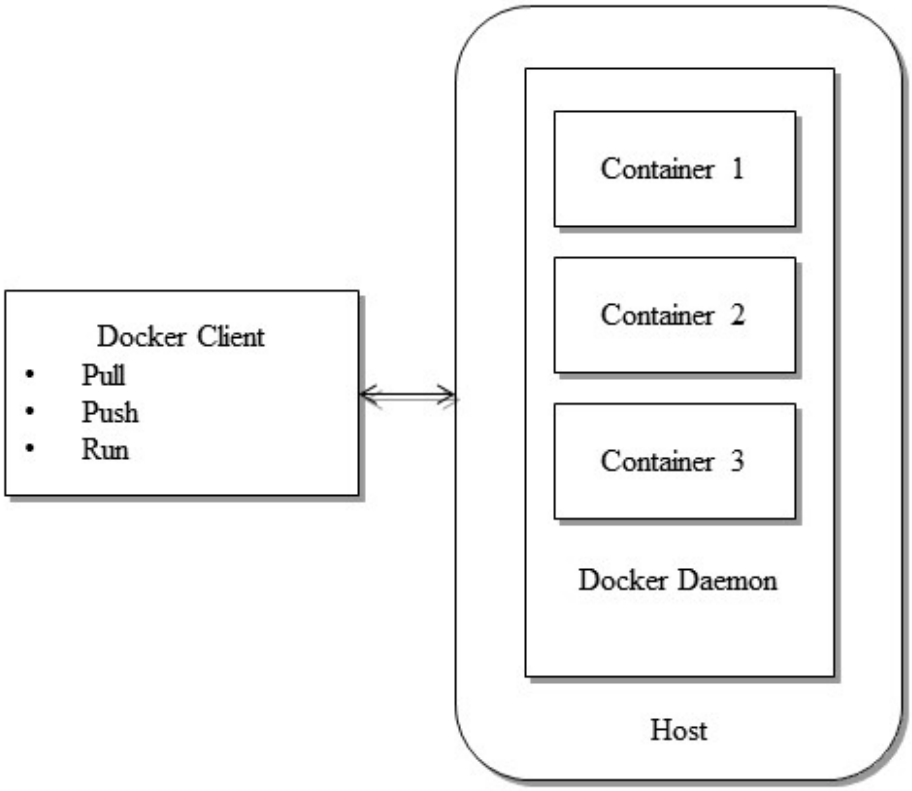

***Figure 5.3**. Docker communication overview between Docker client and daemon.*

Docker technology utilizes several Linux kernel features [14] from which them main features are:

- Chroot (change root) is a Linux command to change the root directory of the current process and its children to a new directory. Some of the containers use chroot to isolate and share the filesystem across virtual environments.
- Cgroups is part of kernel subsystem, provides a fine grained control over sharing of resources like CPU, memory, group of processes etc. Almost all the containers use cgroups as the underlying mechanism for resource management. Cgroups set the limits on the usage of these resources.
- Namespaces create barriers between containers and provide isolation between processes. When docker container is started it creates set of namespaces and cgroups for the container. Following namespaces are used.
    - PID namespace provides process identifiers to a list of processes.
    - NET namespace allows each container to have its network artifacts like routing table and iptables.
    - IPC namespace provides isolation for various IPC mechanisms namely semaphore, message queues and shared memory segments.
    - MNT namespace provides container with its own mountpoint.

- UTS namespace ensures that different containers can view and change hostnames specifically to them.

Docker client provides a command line tool to inspect containers' resource consumption. Docker stats monitors for CPU utilization for each container, memory used, total memory available and total data sent and received over the network by the container. This information can be used to identify possible performance problems during testing. Default monitoring client does not provide ability for alarms but additional plugins can be installed for alarms.

5.6 Docker Images

A Docker image is a collection of all of the files that make up a software application. Docker uses union file system to combine different parts of the system into a single image. Union file systems allow files and directories of separate file systems, known as branches, to be transparently overlaid, forming a single coherent file system. A Docker image can build on top of another Docker image that builds on top of another Docker image and so on. The first image layer is called a base image, and all other layers except the last image layer are called parent images. They inherit all the properties and settings of their parent images and add their own configuration in the Docker file. [54]

When a new change is made to a Docker image, Docker will add a new layer to the image. Now instead of replacing the whole image or entirely rebuilding, as with a virtual machine, only that layer is added or updated. Docker containers are built from these base images by using a simple, descriptive set of instructions. Each instruction creates a new layer in the image. Docker reads this Docker file, executes the instructions, and returns a final image of the system. Common instructions used in Docker are [12]:

- Run a command.
- Add a file or directory.
- Create an environment variable.
- What process to run when launching a container from this image?

Docker provides developers with lightweight abstraction, better performance than VMs, and easy application portability. Combining Docker with VMs is one option that brings the best of both worlds. Docker is Linux centric technology and runs only on Linux, this can be leveraged with the use of virtualized infrastructure that runs Linux system [12]. Main challenges with Docker are [25]:

- Security. The Docker engine dependents on the host operating system for security, which means that the developer needs to ensure resource isolation and access control. User needs to ensure that the right personnel have access to the execution

environment. Process control isolates the applications from interfering with one another.

- Scalability. Large productions infrastructures have large application stacks to which Docker needs to be compatible. Also applications running inside Docker containers don't store data within the file system and once container is shutdown the data is lost. This data needs to be backed up with recovery mechanisms. Building large container images for large applications still takes time and

Docker won't be implemented as a part of the deployment framework. This is due to two reasons. First, Docker is pretty new technology and getting reference help can be difficult performance in large production environments is unclear. Second, Docker relies heavily on Linux environment and can have issues running in large heterogeneous production environment. If Docker were to be implemented, it would be used to set for testing the application where tests would be done simultaneously in separate containers. Docker be used for its speed and flexibility during development by rapidly creating different environments. The advantages of Docker do not necessarily outweigh the opportunity cost of rewriting the company' entire IT infrastructure. Docker is suitable for enterprises that build their infrastructure from ground up. For this thesis Docker was showcased as an alternative virtualization technology to implement in businesses infrastructure.

6. DEPLOYING SOFTWARE WITH ZERO DOWNTIME

This chapter will focus on software deployment process. Based on the research, a deployment framework is presented. The framework is used for software deployment and it utilizes deployment pipeline, Puppet tool and virtualization. Overview of these techniques was given in previous chapters. First, an overview of two zero-downtime deployment methods: blue-green deployments and canary release, is presented. Next a deployment framework is proposed and explained. Evaluation of the framework is given at the end of the chapter.

6.1 Zero- Downtime deployments

6.1.1 Blue Green deployments

Idea behind blue-green deployments is to have additional, completely identical, operating environments in standby that will become as a new environment. First environment is serving as live environment while the second environment acts as a staging environment for the new software version. Team installs new software to the second environment and performs all necessary tests to make sure it is working properly. Once satisfied with its stability all users are then switched from current production environment into a newly created environment. Switching users happens by changing the router configurations so that all new incoming requests come into new environment. This switchover happens in seconds.

During switchover process the users may have some data lost as it takes time to migrate data between databases. One solution to this is to have database migration done separately from application deployment and perform the upgrade independently. Another solution is by putting application into read-only mode, synchronize this database to new environment, and then perform deployment process after which application is put back to read-write mode. Once performance of the new production environment is stable, operations team brings down previous production environment route users to second environment. This standby environment becomes now primary production environment while the previous one is idle. [17] Figure 6.1 illustrates this idea.

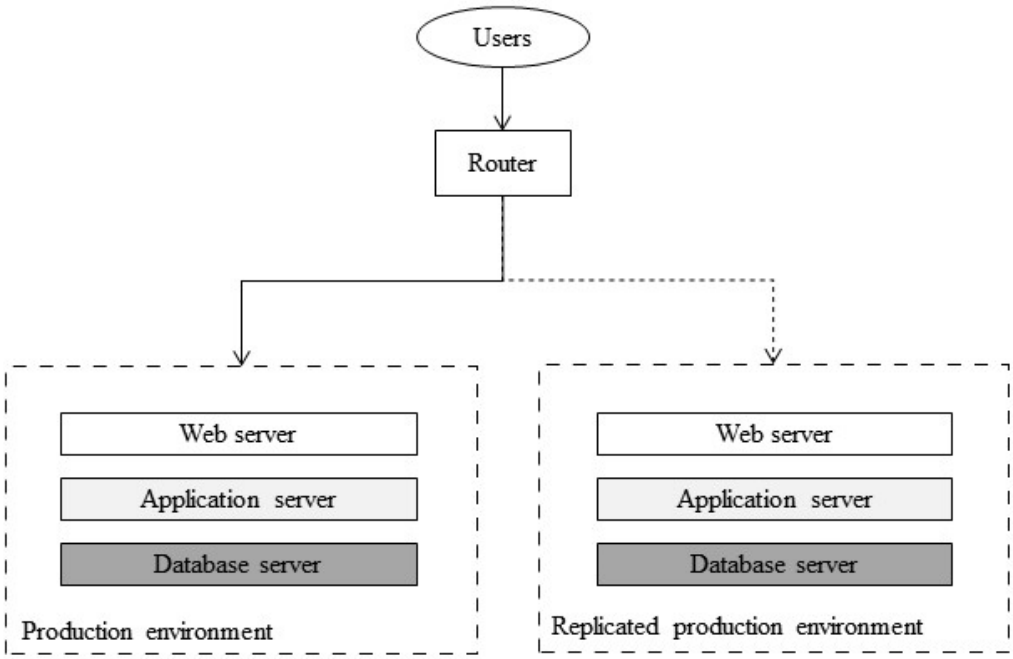

Figure 6.1. *Blue-green deployment method where all users are switched from one production environment into a new upgraded production environment.*

Both environments are completely separate inside their own networks with own IP identities. Each environment is operating on the same or different hardware as long as they are partitioned in different zones with separate IP addresses. Changing environment configurations will not affect other environments functionality in any way. [2] Both environments need have same configurations so that during switching the operation stays the same way. Advantage of using this deployment method is that it makes rollbacks easy. If something goes wrong with the new deployment just rollback to previous environment, which is in good state and ready. [17]

6.1.2 Canary release

Canary releasing method uses additional environment for production just like in blue-green deployment. Instead of switching all users to the new environment, only a small part of users are switched. This idea is illustrated in Figure 6.2. Users are routed to another environment where the stability of new software is tested under actual load. Canary method can be used if the success of the new deployment is not guaranteed. Furthermore it makes debugging the deployment process easier as it provides fast feedback on the new features. If performance of a new environment does not meet the requirements, then users are switched back to the previous environment. Users are incrementally routed to the new environment, ensuring that no data is lost and that the performance doesn’t suffer until all users are using the new environment.

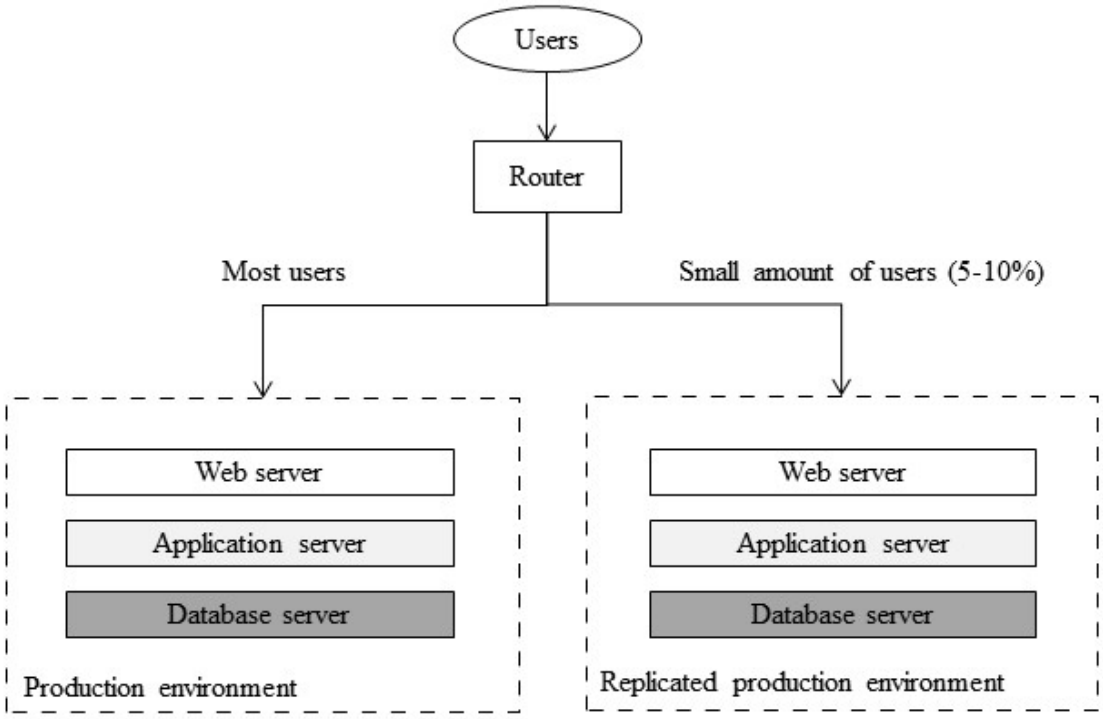

Figure 6.2. *Canary release where a small test set of users are switched from one production into a new upgraded production environment.*

Having two production sites up and running at the same time can be painful as existing resources need to work with both versions of the application. Data synchronization needs to be fast to avoid data corruption if the users are switched between environments. Benefits for using canary releases are [2]:

- Rolling back is easier than in blue green deployment. Only a handful of users are switched to new environment instead of all at once.
- Testing new environments. Perform smoke tests, capacity tests to the new version as more users are routed and verify that it performs well. Performance tests are more reliable as software monitors applications metrics with added user load. Different metrics to monitor include CPU usage, I/O, and memory usage. For large production environments this is only reasonable way of conducting the performance tests.
- A/B testing. Canary releases can be used to test new features of the software to get real user feedback and deciding whether new features need to be implemented.

6.2 Requirements for deployment setup

Based on the research done at Nokia the following requirements are identified for the software deployment framework.

- Horizontally scalable. The system needs to handle addition of new nodes and have a way to quickly provision them. Performance of the system should not be degraded when new nodes are added.
- Reliable. Once the application is ready for deployment, it needs to be sure that new application can handle production environment needs and possible unexpected scenarios eliminated. Feedback needs to be delivered at the end of each stage.
- Fault Tolerant. System need to be able to report possible defects to responsible teams as soon as possible. Also the system needs to store generated reports in centralized repository.

6.3 Proposed deployment framework

Figure 6.3 presents proposed software delivery framework that utilizes deployment pipeline, Puppet tool and hypervisor based virtualization. The pipeline presented consists of five stages: Commit, Acceptance, User, Capacity and Production stage. The process starts with developers committing changes into version control system. Continuous integration server, Jenkins, monitors for changes and starts a new pipeline instance if it detects new commits. Jenkins will automatically trigger a new instance in the pipeline where the new code is propagated.

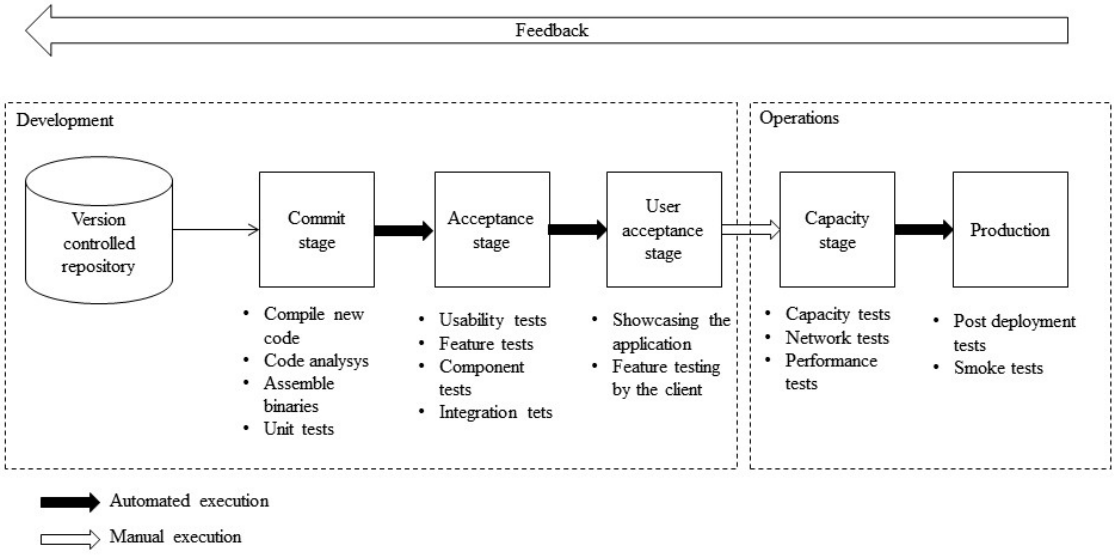

Figure 6.3. *Proposed deployment pipeline for software delivery.*

Before application is released to production it must pass the following stages.

- Commit stage. Commit stage begins the pipeline. Jenkins monitors possible changes done to the code are compiled, unit tests performed, code and syntax analysis and installers created. The outcome of commit stage are binaries from the source code.
- Acceptance stage. Jenkins triggers the build into acceptance tests after successful commit stage. This stage serves as testing the environment for the application. This stage is heavily focused with automation to reduce cycle time, as this stage is the most manually intensive if different scenarios are performed manually.
- User acceptance stage. Once developer is confident with the application it is showcased to the business analyst to verify the user requirements.
- Capacity stage. This stage tests the systems performance with simulated traffic. Capacity testing environment acts as a staging environment for the application. Once the application is verified to be production ready it is turned into live environment by routing real users into it. This stage is manually triggered from acceptance test environment. Capacity stage acts as a quality assurance as this is the last stage before release and it ensures that the application meets the requirements of the customer. This stage is most important stage in this pipeline as it ensures that the build is fit for the new production environment, it can handle the user load and checks all previous stages in the pipeline.

In this framework VMware is used to create VMs that will serve as testing environments. Linux operating system is installed after the VM is provisioned. Next Puppet tool is installed. Once Puppet is installed on a VM, the client will connect to a master node proceeded by downloading and applying configurations to that node. Puppet handles installation tasks by first pulling the build from repository, followed by installing and configuring the package. Post installation tasks include smoke tests that verify that the software can be started and it communicates with different environments. At the end of each pipeline stage Jenkins will send a result report to repository.

Current framework is not an optimized solution for deployments but acts more of a starting point from which teams start optimizing it. Deployment pipeline follows a linear pattern which is not optimal as it takes the most time to execute. Issues may occur once certain stages take too long to execute thus prolonging other stages. Such possible bottleneck can happen with acceptance stage as it takes usually the longest to complete. This is problematic in fault tolerant cases where setting up new emergency environment would take much time. Solution to this would be running tests in parallel so that deployment ready versions are pushed out quickly. In this framework both acceptance stages could run parallel as well as capacity testing and releasing to production stages.

Parallelization was not included in this design because of reliability and feedback requirements. When deploying for the first time, teams need to get optimized feedback on different stages of the pipeline. Increased reliability is done at the cost of deployment time. Once the deployment process gets reliable enough certain stages could be executed in parallel. Optimized solution for reduced deployment time would be running all automated stages of the pipeline is parallel. In this case commit, acceptance and user acceptance tests would be done in parallel, then manually moved on to operations environment where capacity and production stages would be done in parallel.

Manual signoff is used between development and operations environment. This is because this stage typically means handing the software to other team that then performs deployment. Manual signoff increases deployment reliability with the collaboration between teams. Possible issue may rise with access control as only certain parties should be able to do the signoff. Also possible bugs and glitches may be introduced between different environments. This may result in blaming each other between teams. Ideally there would be no specific teams instead all responsible parties collaborate with each other for increased efficiency.

6.3.1 Provisioning environments

Puppet is used to provision new environments. New VMs are created by copying existing VM images. These images have operating system and Puppet preinstalled inside. New VM is created by cloning existing ones. Puppet downloads and installs necessary software packages, configures them and enables necessary services. Hypervisor sets necessary information about the VM including host id and IP address.

Puppet master is installed in development environment with necessary manifest files stored in own directory. Following procedures are carried out when installing agent nodes. First Puppet is installed to the system and connection established to the master node. Establishing connection happens through command line in following manner [8]:

1. puppet agent --server=dev.master.com. The --server part specifies the name of Puppet master node which in this case is dev.master.com. Agent client sends a request that master node accepts.
2. puppet cert --sign test.node.com. Puppet master completes and authenticates by signing the certificate to out agent node ‘test.node.com’. Puppet master can now propagate changes to the client.

Manifests are stored inside Puppet master’s directory /etc/puppetlabs/puppet/modules, from which they are distributed to agent nodes. Two types of configurations are used to provision VMs. First type of manifests describe what software packages are installed

alongside necessary configurations of files and services associated with the software package. Listing 6.1 shows this type of configuration module.

```
class ssh {
     package { "openssh":
          ensure => present,
     }
     file { "/etc/ssh/sshd_config":
          ensure = > present,
          owner => 'root',
          group => 'root',
          mode => 0600,
          source => "puppet:///modules/ssh/sshd_config",
     }
     service { "sshd":
          ensure => running,
          hasstatus => true,
          hasrestart => true,
          enable => true,
     }
}
```

Listing 6.1. *Class module for managing ssh service.*

Above module manages ssh resource on a node. The module declares that openssh package needs to be installed, configures ssh file to the user and specifies that sshd service needs to be running. Other Puppet modules follow this same pattern to configure their resources. Second type of configurations describe which packages are installed to specified nodes. Listing 6.2 shows an example manifest, nodes.pp, that specifies configurations to different nodes.

```
# /etc/puppet/manifests/nodes.pp

class base {
     include postfix
     include ssh
     include python
}
node 'acceptance.dev.com' {
     include base
     include acceptance
}
node 'user.dev.com' {
     include base
     include user
}
node 'capacity.dev.com' {
     include base
     include capacity
}
```

Listing 6.2. *Example module used to specify necessary resources for different nodes.*

Above listing declares which resources are installed to selected nodes. `Base` resource is deployed to all nodes. It contains configuration for services, files needed to run the software. Other listed resources create environment specific VMs that are used at different stages of the pipeline. Each resource follows the general format as in listing 6.1.

6.3.2 Running tests

The commit test compiles the code into binaries and produces reports. These binaries are tested and then propagated along the pipeline. Commit stage consists of the following steps:

- Compiling the code.
- Create binaries.
- Perform analysis of the code

Commit stage is the first step where code errors are checked and feedback is sent to developers if criteria's are not met. At the end of successful commit stage generated binaries and reports are saved in repository ready for next stage in the pipeline. Created binaries are deployed into production-like environment where different tests are carried out. Puppet is responsible for deploying binaries to target environment. First, virtual environment is created by VMware. Next, Puppet installs binaries to this environment, runs necessary

smoke tests. Then acceptance tests are performed. Each acceptance test executes a certain function in the software and verifies the results. This stage is the most time consuming as the tests are typically performed manually since writing a script for every application build may be time consuming. If a build fails acceptance test then it is discarded from the pipeline and returned to development environment. Objective in this stage is to fail fast, meaning that teams try to locate possible defects as soon as possible and fix them. For user acceptance test there are business analyst and developers. Business analysts defines what requirements are set to the software and reports them to developer. Once developer has performed acceptance tests and verified that they work then this build is then demonstrated to analysts and the customer [2].

Once the build passes user acceptance test it will be manually promoted to last stage where capacity tests are done. Again Puppet is used to provision this capacity test environment. At this stage the configurations for testing environment need to replicate real production environment as this is last stage before deploying the application. Again objective of capacity testing is to fail fast by discovering possible performance issues, reporting and fixing them. Capacity tests are carried out by simulating user traffic to the server and recording these interactions. During capacity test the following metrics are monitored [2]:

- Load tests. How the system handles production-like capacity with increasing number of requests made to the system. This includes the maximum number of transactions the system can handle.
- Latency. How long does it take to resolve each response and how this metric depend on the number of users.

Releasing to production will happen with canary release method. Canary release method was selected because the deployment framework is pretty general and its reliability is not guaranteed. Also it is more flexible than blue-green method since it affects a small number of users making it ideal for testing deployments. Using canary release method gives more confidence in systems stability as load the load increases in the system. This method allows to incrementally synchronize data between different databases, thus minimizing data losses. If something goes wrong during the switchover process the impact is minimized by routing the users to previous environment. Switchover is done my mapping incoming new requests to the new environment.

6.4 Discussion

Developed framework was evaluated based on the literature data. Evaluation was based on the requirements presented in Chapter 6.3 Proposed framework reduces downtime by eliminating the sources of downtime that were presented in Chapter 2. The frameworks

main goal was to minimize manual work and automate major many steps during deployment process. Different stages of the pipeline are orchestrated automatically by performing each stage automatically. Manual steps include provisioning the environments for the first time and signing off the application between different teams. All tests are performed automatically in the pipeline. This will require more planning on the test infrastructure and maintenance of the test scripts compared to manual testing. Current framework may suffer from unexpected bugs that are missed during automated test. If manual tests were to be implemented in the current pipeline, they would be carried out in the later stages of the pipeline. Getting closer to releasing means that more reliability is needed of the application. Capacity testing is the last stage before releasing and that stage could have a mixture of automated and manual testing. Performance test can be carried out manually by monitoring the state of the application under load.

Virtualization was selected to create new environments. Virtualization reduces amount of manual steps when configuring and installing the software. Once a VM is configured for the first time, the image of this VM is saved and stored in a version controlled repository. During new installation this preconfigured image is booted inside a new VM and the operating environment is running in minutes compared to hours if done manually. This can cut the potential downtime from one half to one quarter, depending on how the system is configured [33]. These images are also used during unplanned downtime or in disaster recovery situations to create new operating environments fast.

Running many VMs inside a single machine creates a single point of failure issue, where if that machine happens to go down then all VMs are shut down. Same issue is associated with the single version control repository as this repository contains the application, VM images and Puppet configurations for the deployment pipeline to work. Currently there is no redundancy at all and if the server fails then lots of data can be lost. In case of disaster scenarios this setup suffers from lack of redundant hardware that contains backups for the repository. The server that is responsible for routing the users between different environments does not have any redundancy right now and the load balancer might become a single point of failure. One major cost consideration is the number of VMs on each host. The downside is the related cost to provision the larger number of required physical servers. A smaller number of VMs is less likely to strain the hardware resources on the host. Decision should be based on a cost-benefit analysis [49].

In this framework two distinctive teams are responsible for delivery process, development and operations teams. This can pose a problem as it can make the developers not being fully committed during the build stage which further results in bad tests later on in the pipeline. For instance, planned changes during the build phase are sometimes not communicated to the operations team and, thus possibly breaking the tests without warning [35].

Presumably development and operations environments are running on own separate hardware's with own virtual environments. Having two separate environments can pose a problem if they differ from another. Components need to be exactly same that environments are consistent on all machines. Typical problem is that the deployment process works on one machine but not the other. The pipeline now uses manual handoff between development and operation teams. Manual handoff was selected to reduce the confusion between different teams on the functionality of the application. Human error was one of the main reasons behind downtime and was commonly result of insufficient knowledge of the application. Manual handoff enforces collaboration between teams thus increasing the reliability of the deployment process. Best way to mitigate downtime is communication between all responsible stakeholders for the pipeline by understanding the "big picture" [35, 36].

Puppet tool was selected because it supports multiple environments such as testing, staging and production and can manage up to ten thousand nodes for added scalability. Still Puppet might not operate at a high enough abstraction level to manage an entire infrastructure. [37], [38]. Currently there are two concerns related to Puppet. First one is that Puppet just declares the state of the machine. How feasible solution would it be to just provision virtual machine to a specific state and then create of copy of that machine? This image would then be used to set up environments. Reasoning for Puppet is that it offers flexibility for environment provisioning. Configurations to environments are done by modifying the manifest file at Puppet master and then pulled by agent nodes. Another issue with Puppet is scalability. Puppet uses client-server model to distribute configurations. There is a limit on how many nodes the master client can serve until the service suffers from increased lead times. Large production environments with thousands of nodes can pose this threat. One solution would be using many Puppet masters at the cost of added complexity. If the master server fails then client nodes cannot be managed until master server is back online.

Proposed high availability mechanism is based on server clusters which can raise the bar for hardware homogeneity. Homogeneity requirements can increase hardware acquisition cost and reduce flexibility in hardware provision. [49] The main challenge with migration is that both source and destination physical hosts need to have access to VM's disks, where the guest OS and data are stored. Shared storage or filesystems are used in local environments, but the approach is not appealing when wide area networks are involved due to performance and connectivity limitations. Also the network configuration of VMs should remain unchanged after migration, which is easier if the migration is performed within a single broadcast domain, but difficult when subnets boundaries are crossed. [53]

External factors may affect service availability, most notably distributed denial of service attacks and malware that are capable of interrupting service. These attacks have serious

effect on routers and load balancers which manage the user traffic as they can cause outages in these devices. Current availability method does not have native tools to address these external factors. Prevention comes from firewalls, intrusion detection systems and virus scanners [49].

7. CONCLUSIONS

The goal of this thesis was to study available tools and techniques that can be used to reduce software deployment time. The study introduced deployment pipeline practice, configuration management tool Puppet, hypervisor based virtualization and Docker tool. Based on the study a deployment framework was developed that utilizes the practices and tools presented in this thesis. Nokia Networks used this study to cross check it with their deployment process and possibly optimize it. Study was carried by studying available public documents and case studies.

Developed deployment framework was used to address research problem of the study. Solution to reduce downtime was proposed by incrementally rolling out the software to users while validating software's stability. Puppet and virtualized environments are used to create consistent environments that mimic real production environments. Motivation behind proposed framework was to minimize the amount of manual work done by executing deployment scripts that perform these steps during deployment. Software downtime during installations is reduced substantially depending on the size of the application, the quality of configuration scripts and architecture of production environment. Planned downtime is reduced by creating operating environments fast, running automated tests, installing and configuring the application to that environment. Built in redundancy and migration service reduce unplanned downtime in the system by monitoring the systems performance. Proposed framework acts more or less as a baseline for deployment process. It is not an optimized solution to any specific environment and needs to be performed several times and optimized based on deployment requirements.

Puppet tool proved to be easy to learn technology and had many references to ease the study. Puppet is a mature technology and is widely used in many corporate environments and proved to be viable solution in this study. Docker tool was introduced as out of the box technology that imposed own advantages and disadvantages for deployment. Docker allows more efficient use of virtual resources and fast deployment. Still LXCs and Docker are promising technologies but using it in large heterogeneous enterprises can be unreliable, due to limited support and reliability and thus was not part of the deployment framework.

Actual implementation of the framework was not part of this study. Future studies can be conducted to evaluate other deployment techniques and solutions. One study proposal is comparing releasing methods, blue-green and canary methods, described in this thesis in more detail. Research can be done around switching process and how rollbacks are implemented. Other research topics include studying feasibility of other tools that can be used for deployment, migration tools for VMs so that service appears to be interrupted to

the client. All major virtualization service providers have high availability mechanisms but they differ. Needless to say Puppet and Docker are one of the many tools which enterprises can use to implement own solution for reduced downtime. Based on this study Table 7.1 sums up the patterns and anti-patterns covered in this thesis for software deployment.

Table 7.1 *Patterns and anti-patterns for software deployment process.*

	Pattern	**Anti-Pattern**
Automated Deployments	All deployment processes are written in a code which is then executed to perform software deployment. Regular collaboration between different teams is needed to ensure that the deployment is applicable.	Manually deploying software. Little collaboration between teams causing problems as teams don't fully understand the code.
Deployment Pipeline	Deployment pipeline is used to automatically validate the software with automated tests and feedback. Pipeline grants the visibility of all the stages that the software must undergo before releasing.	Not implementing feedback on the software build. Manually triggering different software build and test stages.
Blue-Green Deployments	Software is deployed to a staging - environment while production continues to run. Once staging environment is validated it is switched as a new production environment.	Production is taken down while new application is installed.
Canary Release	Software is released to production for a small subset of users to get feedback prior to a complete rollout.	New software is released to all users at once, thus making rollbacks harder.
Monitoring	Monitoring different metrics in development and production with built it in alarms. Monitoring deployment pipeline to find possible errors fast.	Not having monitoring software or monitoring is carried out manually.
Virtualization	Emulate hardware resources and create environments to test and simulate the software. Ensure that environments are consistent.	All development is done on separate machines thus causing inconsistency between environments.
Configuration management	Using configuration management tools to manage infrastructure and to automatically deploy software packages.	Configurations are done manually or infrequent use of configuration tools.
Redundancy	Having additional hardware for mission critical components in deployment system so that in case of outages service can be moved to another host.	No redundant hardware is used thus posing single links of failure.

REFERENCES

[1] E. Marcus, H. Stern, “Blueprints for High Availability”, Wiley Publishing, 2003.

[2] J. Humble, D. Farley, “Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation”, Addison-Wesley Professional, 2010.

[3] D. Cotroneo, R. Natella, R. Pietrantuono, S. Russo, “Software Aging Analysis of the Linux Operating System”, Software Reliability Engineering (ISSRE), 2010.

[4] S. Cummins, “Pro SharePoint Disaster Recovery and High Availability”, Apress, 2013.

[5] D. Spinellis, "Don't Install Software by Hand", IEEE Software, vol.29, no. 4, pp. 86-87, 2012.

[6] P. Duvall, “Continuous Delivery Patterns and Antipatterns in the Software Lifecycle”, [WWW], Available (accessed on 28.4.2015): http://ref-cardz.dzone.com/refcardz/continuous-delivery-patterns

[7] J. Loope, “Managing Infrastructure with puppet”, O’Reilly Media, 2011.

[8] J. Turnbull, J. McCune, “Pro puppet“, Apress, 2011.

[9] B. Wootton , “Preparing for continuous delivery”, DZone Inc, .

[10] Jenkins, “Meet Jenkins”, [WWW], Available (accessed on 6.5.2015): https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins.

[11] Puppet Labs, “Puppet Documentation Index”, [WWW], Available (accessed on 1.5.2015): http://docs.puppetlabs.com/puppet/.

[12] Docker Inc, “About Docker”, [WWW], Available (accessed on 1.5.2015): https://docs.docker.com/.

[13] S. Holla, “Orchestrating docker”, Packt Publishing, 2015.

[14] R. Dua, R. Raja, D. Kakadia, “Virtualization vs Containerization to support Paas”, 2014 IEEE International Conference on Cloud Engineering, 2014.

[15] J. Arundel, “Puppet 3 Beginners Guide”, Packt Publishing, 2013.

[16] M. Belguidoum, F. Dagnat “Dependency Management in Software Component Deployment”, Electronic Notes in Theoretical Computer Science, 2007

[17] M. Fowler, “BlueGreenDeployment”, [WWW], Available (accessed on 29.4.2015): http://martinfowler.com/bliki/BlueGreenDeployment.html.

[18] M. Fowler, “Continuous Integration”, [WWW], Available (accessed on 29.4.2015): http://www.martinfowler.com/articles/continuousIntegration.html.

[19] M. Cohen, K. Hurley, P. Newson, “Google Compute Engine Managing Secure and Scalable Cloud Computing”, O'Reilly Media, 2014.

[20] M. Peacock, “Creating Development Environments with Vagrant”, Packt Publishing, 2013.

[21] Dzone, “Guide to Continuous Delivery”, 2015.

[22] M. Huttermann, “Devops For developers”, Appress, 2012.

[23] OpenNMS, “About OpenNMS”, [WWW], Available (accessed on 14.5.2015): http://www.opennms.org/about/.

[24] J. Arundel, “Puppet 3 Cookbook”, Packt Publishing, 2013.

[25] B. Golden, “Docker and Its Challenges for Enterprise IT”, [WWW], Available (accessed on 16.5.2015): http://www.activestate.com/blog/2015/02/docker-and-its-challenges-enterprise-it.

[26] M. Cataldo, A. Mockus, J. Roberts, J. D. Herbsleb, “Software Dependencies, Work Dependencies, and Their Impact on Failures”, IEEE Transactions on Software Engineering, 2009.

[27] M. Hayun, “Continuous Integration & Package Management 101”, [WWW], Available (accessed on 14.5.2015): http://www.slideshare.net/maorhayun/continuous-integration-package-management-101-33131956

[28] W. Hagen, “Professional Xen Virtualization”, Wrox Press, 2008.

[29] “Assessing the Financial Impact of Downtime”, [WWW], Available (accessed on 18.5.2015): http://www.strategiccompanies.com/pdfs/Assessing%20the%20Financial%20Impact%20of%20Downtime.pdf

[30] W. Fischer, “IPMI Basics”, [WWW], Available (accessed on 18.5.2015): https://www.thomas-krenn.com/en/wiki/IPMI_Basics#Primary_IPMI_Features

[31] Puppet Labs, “Reporting”, [WWW], Available (accessed on 18.5.2015): https://docs.puppetlabs.com/guides/reporting.html

[32] S.Vaughan-Nichols, “Why Containers Instead of Hypervisors?“, [WWW], Available (accessed on 18.5.2015): http://blog.smartbear.com/web-monitoring/why-containers-instead-of-hypervisors/

[33] Jacob Haugen, “Improving Downtime and Disaster Recovery with Virtualization”, [WWW], Available (accessed 18.7.2015): http://www.mbtmag.com/articles/2015/06/improving-downtime-and-disaster-recovery-virtualization

[34] Trefis Team, “Growing Competition For VMware In Virtualization Market”, [WWW], Available (accessed 18.7.2015): http://www.trefis.com/stock/vmw/articles/221206/growing-competition-for-vmware-in-virtualization-market/2014-01-07

[35] J. Gmeiner, R. Ramler. J. Haslinger, “Automated Testing in the Continuous Delivery Pipeline: A Case Study of an Online Company”, IEEE Eighth International Conference on Software Testing, 2015.

[36] S. J. Bigelow, “The causes and costs of data center system downtime: Advisory Board Q&A“, [WWW], Available (accessed 18.7.2015): http://searchdatacenter.techtarget.com/feature/The-causes-and-costs-of-data-center-system-downtime-Advisory-Board-QA

[37] B. Vanbrabant, W. Joosen, “A framework for integrated configuration management tools”, IFIP/IEEE International Symposium on Integrated Network Management, 2013.

[38] T. Delaet, W. Joosen, “A survey of system configuration tools”, [WWW], Available (accessed 19.7.2015): https://www.usenix.org/legacy/events/lisa10/tech/full_papers/Delaet.pdf?CFID=694697875&CFTOKEN=74147045

[39] K. Schmidt, “High Availability and Disaster Recovery Concepts, Design, Implementation”, Springer Publishing, 2006.

[40] S. Pippal, S. Singh, R.K. Sachan, D.S. Kushwaha, “High availability of databases for cloud”, 2nd International Conference on Computing for Sustainable Global Development (INDIACom), 2015.

[41] C. Janssen, “Single Point of Failure (SPOF)”, [WWW], Available (accessed 19.7.2015): http://www.techopedia.com/definition/4351/single-point-of-failure-spof

[42] E. Bauer, “Design for Reliability: Information and Computer-Based Systems”, Wiley-IEEE Press, 2010.

[43] E. Katihar, F. Khendek, M. Toeroe, “Operating system upgrade in high availability environment”, International Conference on Software, Telecommunications and Computer Networks (SoftCOM), 2013.

[44] A. Kankkunen, “Non-service affecting software upgrades for multi-service routers”, Proceedings.5th International Workshop on Design of Reliable Communication Networks, 2005.

[45] S. Crouch, “Defending your code against dependency problems”, [WWW], Available (accessed 10.8.2015) http://software.ac.uk/resources/guides/defending-your-code-against-dependency-problems

[46] K. Hoste, J. Timmerman, A. Georges, “EasyBuild: Building Software with Ease”, Companion:High Performance Computing, Networking, Storage and Analysis (SCC), 2012.

[47] A. Kanso, F. Khendek, A. Mishra, M. Toeroe, “Integrating Legacy Applications for High Availability: A Case Study”, IEEE 13th International Symposium on High-Assurance Systems Engineering (HASE), 2011.

[48] Aruba Networks, “High Availability Deployment Mode”, [WWW], Available (accessed 10.9.2015): http://www.arubanetworks.com/techdocs/ArubaOS_64_Web_Help/Content/ArubaFrameStyles/VRRP/HighAvOverView.htm

[49] M. Yang; Y. Wu, “Using Virtualization to Ensure Uninterrupted Access to Software Applications for Financial Services Firms”, 45th Hawaii International Conference on System Science (HICSS), 2012.

[50] “Software Defined Boden, Boden's Cloud Computing, Virtualization and Software Solutions web log“, [WWW], Available (accessed 10.9.2015): http://bodenr.blogspot.fi/2014/05/kvm-and-docker-lxc-benchmarking-with.html

[51] VMware, Inc. “VMware VMotion Live Migration for Virtual Machines Without Service“, [WWW], Available (accessed 18.9.2015): http://www.vmware.com/files/pdf/VMware-VMotion-DS-EN.pdf

[52] VMware, Inc. “VMware Distributed Resource Scheduler (DRS) Dynamic Load Balancing and Resource Allocation for Virtual Machines“, [WWW], Available (accessed 18.9.2015): http://www.vmware.com/files/pdf/VMware-Distributed-Resource-Scheduler-DRS-DS-EN.pdf

[53] M. Tsugawa, R. Figueiredo, J. Fortes, T. Hirofuchi, H. Nakada, R. Takano, “On the use of virtualization technologies to support uninterrupted IT services: A case study with lessons learned from the Great East Japan Earthquake”, IEEE International Conference on Communications (ICC), 2012.

[54] O. Hane, “Build Your Own PaaS with Docker Create, modify, and run your own PaaS with modularized containers using Docker”, Packt Publishing, 2015.