CONTINUOUS INTEGRATION AND CONTINUOUS DELIVERY PIPELINE AUTOMATION FOR AGILE SOFTWARE PROJECT MANAGEMENT

Indunil Suriya Arachchi

(148204F)

Degree of Master of Science

Department of Computer Science and Engineering

University of Moratuwa
Sri Lanka

April 2018

CONTINUOUS INTEGRATION AND CONTINUOUS DELIVERY PIPELINE AUTOMATION FOR AGILE SOFTWARE PROJECT MANAGEMENT

Suriya Arachchige Indunil Bandara Suriya Arachchi

(148204F)

Thesis submitted in partial fulfillment of the requirements for the degree Master of Science

Department of Computer Science and Engineering

University of Moratuwa
Sri Lanka

April 2018

DECLARATION

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

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

Signature: Date:

The above candidate has carried out research for the Masters thesis under my supervision.

Signature of the Supervisor: Date:

ABSTRACT

Adaptation of agile methodologies in software development life cycle has proved an improvement in productivity and quality of systems. In terms of quality, it defines new process and standards requirement where Continuous Integration (CI) principles have filled the gap while improving the quality of system continuously and Continuous Delivery (CD) approach has made faster delivery of software. Continuous Deployment extends the CD features and delivers the software to the production through automation by completing the pipeline. Ultimately, the Continuous Integration Continuous Delivery (CICD) pipeline approach has increased the efficiency and the productivity of agile software projects.

In agile, new features are introduced to system in each sprint delivery, and although it is well developed, the delivery failures are inevitable due to performance issues. By considering delivery timeline, moving for system scaling is common solution in such situations. But, how much system should be scaled? System scale requires current system benchmark status, and expected system status. Benchmarking the production is a critical task, as it may interrupt the live system, which may causes system unstable. New software version should go through a load test, to measure expected system status. The traditional load test methods are unable to identify production performance behavior due to simulated traffic patterns are highly deviated from production.

To overcome those issues, this approach has extended CICD pipeline to having three phase automations process named benchmark, load test and scaling. It minimizes the system interruption by using test bench approach when system benchmarking and it uses the production traffic for load testing which gives more accurate results. Once benchmark and load test phases completed, system scaling can be evaluated. Test bench setup was done on high capacity computer using Ansible automation which provisioned local virtual instances for application servers, Nagios service and load balancing. A simple XML based application which processes cached data by reading files is used to reduce the complexity of test bench approach. Initially, the pipeline was developed using Jenkins CI server, Git repository and Nexus repository with Ansible automation. Then GoReplay is used for traffic duplication from production to test bench environment. Nagios monitoring is used to analyze the system behavior in each phase and the result of test bench has proven that scaling is capable to handle the same load while changing the application software, but it doesn't optimize response time of application at significant level and it helps to reduce the risk of application deployment by integrating this three phase approach as CICD automation extended feature. Thereby the research provides effective way to manage Agile based CICD projects.

Keywords: Continuous Integration, Continuous Delivery, Agile Manifesto, Version Control System, Configuration Management

ACKNOWLEDGEMENT

First I would like to express my earnest gratitude to my supervisor Dr. Indika Perera for the supervision and advice given throughout to make this research a success.

Then, I would like to thank my family members for the support and encouragement that they have given to me.

Finally, I am grateful to my MSc 2014 batch mates and various online community members who supported me during the Research.

S.A.I.B Suriya Arachchi

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviation	Description
SDLC	Software Development Life Cycle
QOS	Quality of Service
QA	Quality Assurance
CICD	Continuous Integration Continuous Delivery
IDE	Integrated Development Environment
XP	Extreme Programming
LSD	Lean Software Development
TDD	Test Driven Development
UI	User Interface
AWS	Amazon Web Services
SLA	Service Level Agreement
SMS	Short Message Service
SNMP	Simple Network Management Protocol
VCS	Version Control System
RC	Release Coordinator
OSS	Open Source Software
GUI	Graphical User Interface
JVM	Java Virtual Machine
DSC	Desired State Configuration
DSL	Domain Specific Language
CLI	Command Line Interface
API	Application Programming Interface
DNS	Domain Name System
YCSB	Yahoo Cloud Serving Benchmark
PXE	Pre-boot Execution Environment
SSH	Secure Shell

CHAPTER 1

INTRODUCTION

1.1 Background

Traditional software development methodologies are not enough to fulfill nowadays business requirements. Unpredictable fast changing market, customers' complex requirements, rapid change of technology and increase of usage of technology on businesses are main challenges that are faced in software industry. Agile methodologies were introduced to face these challenges and to increase productivity while maintaining the Quality of Service.

Agile practices enable flexibility, efficiency and speed of Software Development Life Cycle (SDLC), which attracted by software development companies [1]. As per Agile manifesto [2], customer satisfaction by early and continuous delivery of software, accepting change requirements even in late development, working software is delivered frequently, daily cooperation between business people and developers are few principles among the twelve principles. It can be Extreme Programming (XP), Scrum, Kanban, Crystal, or Lean Software Development (LSD), or Feature-Driven Development (FDD), all these methods enables customer involvement, collaboration, and client feedback which improve the QoS. Implementation of Continuous Integration and Continuous Delivery (CICD) pipeline on agile has enabled fast delivery of software [3] and increase the productivity.

In year 2000, Martin Fowler [4] presented the idea of Continuous Integration (CI), and later J.Humble and D.Farley [5] extended these ideas into the approach of Continuous Delivery (CD) as a concept of Deployment pipeline. Maintain the single source repository, automate the build process and testing, commit on main branch every day, fix broken build immediately, fast build and easy access of latest executable, transparency and automated deployment are the key CI practices. Main benefit of the CI is reducing the risk as everyone is aware on what is happening and

as there are no longer integrations, it is easy to identify bugs and ultimately leads to less bug software. CI tries to make software bug free and reliable which removes the barrier of frequent delivery. Code commit, build, acceptance test, performance test, manual test, and release to production are example phases of CD pipeline, and Accelerated time to market, Improve product quality, improved customer satisfaction, reliable release, improved productivity and efficiency are key benefits [6] which motivates companies to invest on CD. Nowadays, many infrastructure-as-a-service (IaaS) providers promote CICD approaches by providing various supporting services like automated provisioning, and system monitoring. Since most of software and mobile developments are deployed on IaaS, CICD has become essential part of the cloud computing.

1.2 Research Problem

Systems which follows agile methodologies and which are large and scaled horizontally, widely use CICD pipeline automation with the target of achieving fast integrations and deployments. The deployment accuracy and reliability of system mainly depends on test automation process that runs through CICD pipeline at testing phase. [7], [8] and [9] discussed gravity of having test automation in this process, and how it impacts when in failure.

A system can be well implemented and may pass holistic test cases, but it may revert to previous version, due to a performance issue which identified after a deployment. The decision of system scale is an immediate solution for the issue, since agile process has more concern on delivery of product in committed deadlines. The current system benchmark level and new software benchmark level are two factors which defines the target system scale size. System benchmarking can cause system unstable, since it has to evaluate on production. Also, new version should go through a load test to aware its target level. But traditional load test simulations are highly deviated from production live traffic patterns, which produces unreliable results. Therefore, how to benchmark the system with minimum impact, and how to perform

load test for more reliable result, and how to perform them in less effort, are the key problems address through this work.

1.3 Research Objectives

The objectives of this research are:

- Understanding the CICD pipeline, by analyzing exist methods, approaches and tools which use in process.
- Find suitable method to benchmark the existing system.
- Suggest mechanism to perform incremental load test by using production traffic with minimum system impact.
- Define a scaling factor which maintains the consistent system performance for latest production release.
- Automate the benchmark process and suggested load test method by integrate in to CICD pipeline.
- Evaluate system performance using monitoring tools.

1.4 Research Overview

This Research work is presented in six chapters. Chapter 1 describes how Continuous Deployment assists on agile methodologies as background of this research work. It also explains motivation of the research and the research contributions, as well as the layout of rest of the chapters. What are the available approaches and similar research works, as well as alternatives, which supported to carry out the research are explained under Literature Review in Chapter 2. Chapter 3 discusses methodologies applied on research work and Chapter 4 elaborates the research approach through designs and implementations. The results have been evaluated in Chapter 5. Finally, Chapter 6 discusses research conclusion and future works, and concluded the document with the references section.

CHAPTER 2

LITERATURE REVIEW

2.1 Agile Software Development to CICD

The organizations which are following agile methodologies such as extreme programming (XP), Scrum, Lean Software Development (LSD) have expected to achieve faster time-to-market, satisfied customers and high quality software [1]. Individuals and interactions over the processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation and responding to change over following a plan are key values in agile. These values are representing whole agile process and it reflects team structure and how it behaves and practical application of these are benefitted in most of the cases. As an example, though having more sophisticated tools, and there is no member who can use it properly in team, it's better to have member who interact and do something without tools. Though documentation is more important in SDLC, there is no point of having it when the software is not working, and well developed working software can elaborate all the content in document at any time. These key values clearly answer to the question of what is agile, but to see how agile works, has defined in agile manifesto with twelve principles. As in Agile manifesto [2], below table 2.1 shows the twelve principles.

Table 2. 1 Agile Principles

Agile Principle	Motivation
Satisfy the customer through early and continuous delivery of valuable software	Enables early customer feedback, ensure right product at early stage, reduce risk, customer satisfaction and trust on team.
Accept changing requirements, even late in development	Compete business with adoptive nature is profitable and productive to customers at business
Deliver working software frequently, with a preference to the	Bug fixes, minor features should deliver to production sooner after properly test, without breaking existing

shorter timescale	system
Business people and developers must work together daily throughout the project	To improve efficiency, avoid failures and improve quality by sharing knowledge and providing useful feedback.
Build projects around motivated individuals. Give them the environment and support, and trust them to get the job done	It is highly productive to utilize motivated members in development. This ensures that they are motivated throughout the project, to increase the productivity.
The most efficient and effective method of conveying information to and within a development team is face-to-face conversation	Face-to-face conversations help team to aware progress of development and identify bottleneck and discuss solution to achieve deadlines.
Working software is the primary measure of progress	When discuss productivity, efficiency, performance or any measurement, the base line is production.
Agile promote sustainable development. Sponsors, developers, and users should be able to maintain a constant pace indefinitely	Reduce the workload of the team, by using tools and automations. Focus on having fix set of scope by sync with customer with development team.
Continuous attention to technical excellence and good design enhances agility	A better architecture and design can make project perfectly fit on to the agile process.
Simplicity--the art of maximizing the amount of work not done--is essential	Clear visibility and understanding on the sprint by reducing unwanted wastes, smart work than hard work. Eg, Test-Driven Development (TDD) as in [7].
The best architectures, requirements, and designs emerge from self-organizing teams	Define a simple best architecture and designs to understand by the team members and visible the quality of work
Regular interval checkpoints for measure team effectiveness and then tune and adjust their behavior.	Having team activities and retrospective at end of each sprint and do necessary tuning and adjustments in team structure are more effective

Though an organization follows agile mythologies and all the principles defined, it does not mean that it has gained or improved productivity at maximum level with using existing technologies, or they are more efficient than other organizations. As in the agile principles, "Satisfy the customer through early and continuous delivery of valuable software**",** how an organization achieve this principle, has traditional

software delivery methods proven enough efficiency, are they provide sufficient integrity to system. That's when approach of CI came and it extended its features with CD and combined to have CICD pipeline, which provided more efficiency and productivity on agile processes.

If a company adopts CICD into their agile development process, it will help to achieve all the mentioned principles. Customer satisfaction by early and continuous delivery of software: Adopting CICD improves the QoS, and it enables the facility of deliver product to customer continuously by gaining their faith on development team, as well as assuring the delivery of product on time. Accept requirement changes, even in late development: CI provides code management, and branching will help to face such situation. Working software is delivered frequently in weeks rather than months: CICD encourages and motivates team to deliver software frequently due to automated builds and deployments, and with CICD practices leading organizations deploying 10s, 100s, or even 1000s of software updates per day [14].

2.2 CICD Pipeline

In year 2000, Martin Fowler [4] presented the idea of Continuous Integration (CI), and later J.Humble and D.Farley [5] extended these ideas into the approach of Continuous Delivery (CD) as concept of Deployment pipeline which is well known in nowadays as CICD pipeline. The figure 2.1 shows the abstract view of CICD pipeline process. When developer commit new code changes into code management tool or repository, CI tools will automatically build those changes, and it will deploy to QA then Staging environments to test. After series of test phases, production ready code will deploy into production environment.

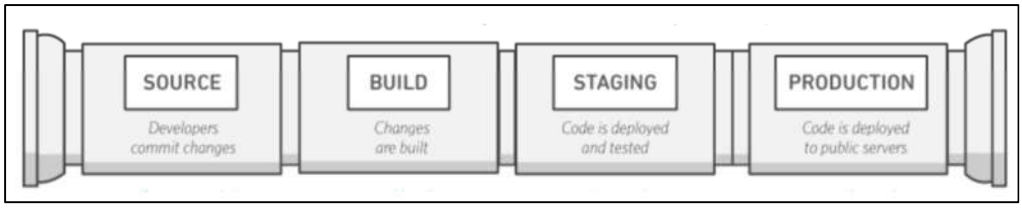


Figure 2. 1 CICD pipeline [13]

In each step of CICD improves the quality of software. When an organization tries to adopt CICD pipeline, they may not be able to adopt it once. First they have to practice CI in organization in order to adopt CD. As in Figure 2.2, having source repository is the first phase that organization can adopt in pipeline. All the development code has to commit into single source code repository, and third party libraries and configurations also can be pushed into repository for common access. Next phase is having CI server and use CI tools for code automatically build and test. Then adopt CD where it pushes build code into relevant environments, and finally the automated deployment by adopting Continuous deployment. The CD is always depending on CI, that's why this process needs to be act as a pipeline.

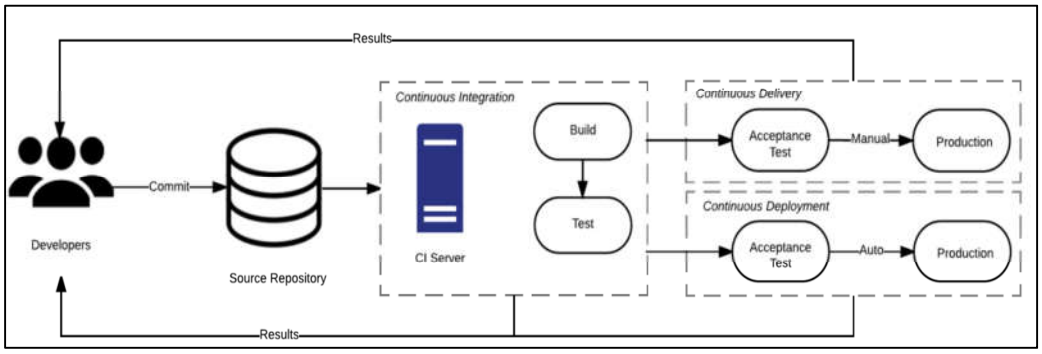


Figure 2. 2 The relation between continuous integration, delivery and deployment [10]

When moving from CI to CD then Continuous Delivery to Continuous Deployment, this pipeline has reduced the manual process execution and finally full process has become automated as in figure 2.3. In CI phase, code change detection, unit testing and perform automated integration testing using build tools and CI tools. Then deployment to testing environments, perform acceptance testing and deploy to production happen in manual. When adopt CD, other than production deployment rest of all the phases done through automation. Finally in Continuous deployment complete process has become automatic and makes the release process faster and make CICD pipeline more efficient.

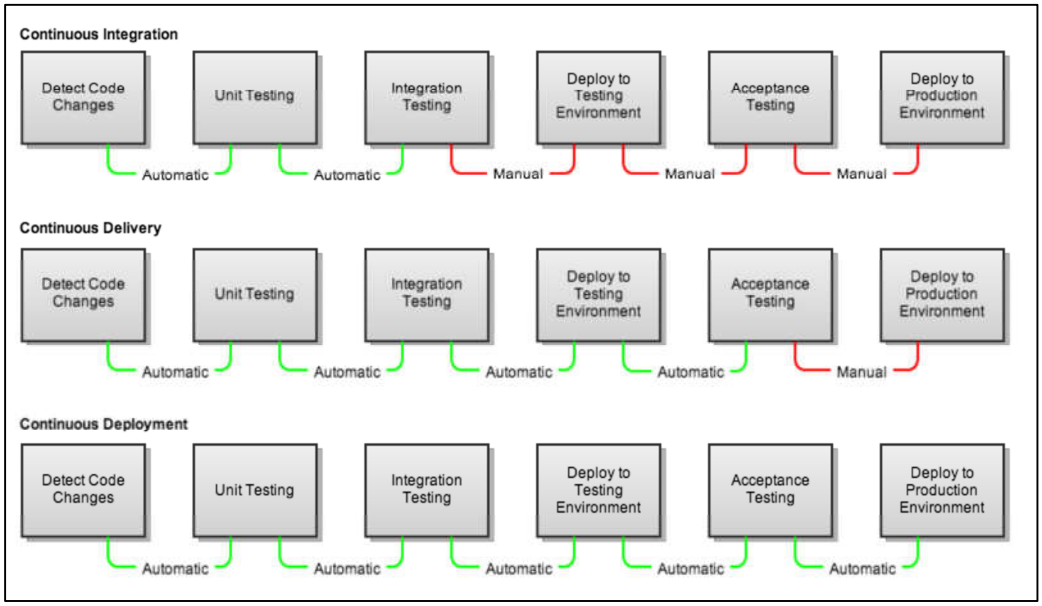


Figure 2. 3 The difference of CI, CD and Continuous Deployment

The main difference between Continuous Delivery and the Continuous deployment is automation at production deployment. In CD more focus on internal process delivery than final deployment. In Some organization there can be several CD processes happen per day, it can be delivery of software to different testing, staging, load-test or benchmark environments, but that final software may deploy to production only once per month. How often it should be delivered and deployed is always dependent on business and application requirement, as some organization does deployment very rarely while others does more frequently, even in hourly.

2.3 Continuous Integration

In 10th September 2000, Martin Fowler [4] presented the idea of Continuous Integration (CI) for the first time, later his work was extended by many research works. CI is a software development practice where team members integrate their work regularly and automate build, test and validate. It helps to find and fix bugs quickly and improve the quality of software. Krusche et al [20] in his research work called Rugby which is agile process model carried out using students has proven that

CI helped to improve the code quality about 50% and it helped to find and fix broken commits faster more than 65% and about 70% of students has claimed that CI helped them to improve overall development workflow.

As in figure 2.4, the source repository and CI server are the main components of the CI. The component source repository includes version control tools like GitLab, Bitbucket and repository like Nexus, Atifactory for keep built executable files as well as for third party libraries. CI server may use external build plugin like ant, maven to build the code and run test cases then display the test and build results. When organization needs to adopt CI as mentioned in CICD they have to iteratively implement it, and they can face many challenges during the process. As per [13], more frequent commits to common codebase, maintaining single source repository, automating builds and automating testing are mentioned as basic challenges and having cloned version of production for testing, visibility of process, allowing easy obtain of any version of application as additional challenges.

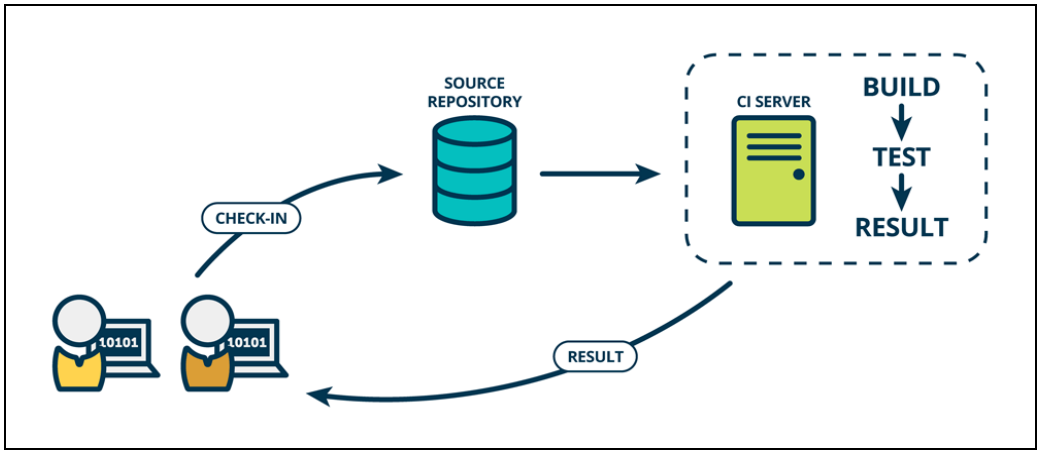


Figure 2. 4 The Continuous Integration [15]

2.3.1 CI Practices

M.Fowler [4] has described how to practice CI in an organization, and how each benefit to progress of development cycle as in table 2.2.

Table 2. 2 CI Practices

CI Practice	Description
Maintain a Single Source Repository	Easy to find source code since all are in one place. Versioning branching and proper naming will resolve different projects and phases
Automate the Build	Automation does, compile source code, file manage and deliver to different environment. Efficient.
Make Your Build Self-Testing	TDD encourages self-testing, in [7] recommend to use TDD prior to requirements being implemented and [9] suggested to use TDD at unit level testing of new development build to make it success of CI with test, build and automation.
Everyone Commits To the Mainline Every Day	Practice to break work load into small chunks which is highly efficient. This CI practice helps developers to improve their communication and early identification of code conflicts and development deviations.
Every Commit Should Build the Mainline on an Integration Machine	Mainline should automatically build periodically (hourly), it alert build failures to fix. Jenkins, Hudson and cruise control support it.
Fix Broken Builds Immediately	Make source code delivery ready always
Keep the Build Fast	Quick response from developers and allow continuous build periodically while gain confident of code.
Test in a Clone of the Production Environment	Clone or shrink down version of production may reduce the risk and save time of testing. (Maintain the same software environment, eg. Using same Java version)
Make it Easy for Anyone to Get the Latest Executable	For others feedback and continuous improvement, easy access through Nexus like repository.
Everyone can see what's happening	Statistics and alerts visible the progress of project, and allow to identify individuals efforts.
Automate Deployment	Automation helps to increase the efficiency of release and it’s more reliable.

2.3.2 CI benefits

Adopting CI process into an organization has given many benefits for software development process. As in [11] following table 2.3 shows the realized benefits by adopting CI practices.

Table 2. 3 CI Benefits

CI Benefit	Description
Build automation	Built automation is more efficient when team does build often, and it helps to identify deployment status soon.
Code stability	Team identifies issues through alerts and notifications, so they can fix them soon.
Analytics	Analytic tools like Sonar dashboard provides stats of breaking of coding standards, to have proper cleaned standard code base.
CD Enablement	Proper branching, source code maintaining and having stable code enables to deliver software always
Faster releases and cost saving	When CD enabled, team can release fast with automation, and it saves time and cost.
Improved productivity and code quality	Automations and having CI practices saves time, so team can utilize in more work, hence productivity increase. Since code is at expected standard, code quality is increased.

2.4 Continuous Delivery (CD)

The idea of CI was extended by J.Humble and D.Farley into an approach of Continuous Delivery, where organizations follow CI easy adopt on CD. According to Humble [5], Continuous Delivery is the ability to get change of all types including new features, configuration changes, bug fixes and experiments into production or into the hands of users, safely and quickly in a sustainable way. Many research works has been carried out on CD and among them [6], [12], [13] and [21-23] are used CD on their research efficiently. In [20], it has introduced CD into multi customer project courses and evaluated its usage, experience and benefits, and in [22] concludes that CD defines “Done” in release process and it enables organizations to become truly agile.

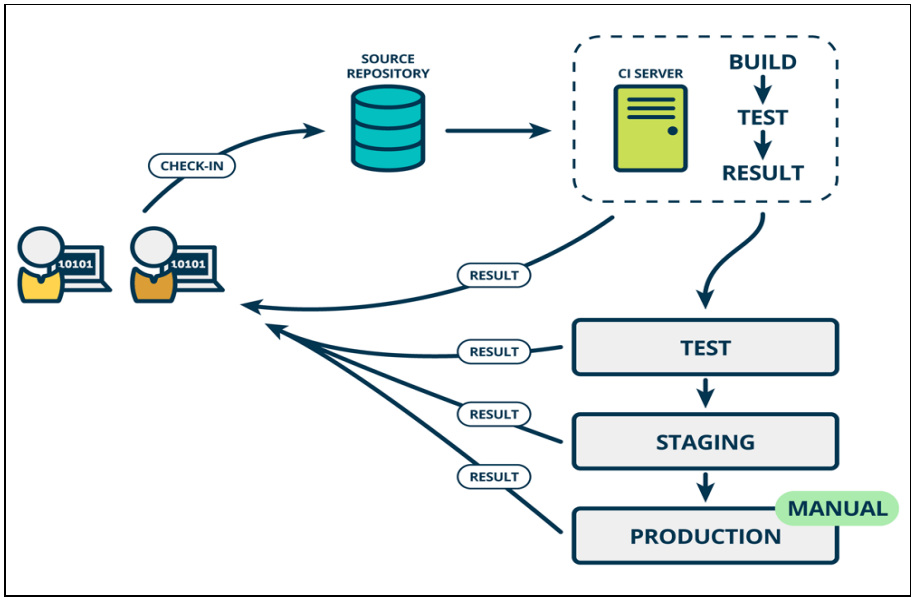


Figure 2. 5 The Continuous Delivery [15]

As shown in figure 2.5, CI is subset of CD, i.e. if organization needs to have CD, they must have CI first, and it also shows that CD is depend on CI server. When CI server automatically build and run automated test scripts as well as analytic scripts if available, it creates executable files. How to deliver these executable files into test, staging and other environments? The process of CD also initiates through CI server, where it executes shell commands and scripts to copy files to necessary environments and do the configuration changes. CD is challenging task, as it has to communicate with different servers, environments and software. The most common issue in adopting CD is permission issues, where those permissions are provided by operational team only.

2.4.1 CD benefits

When adopting CD approach into an organization, they get huge benefits [13], [20] - [22]. As per Chen [6] there is a trend of increasingly invest on CD by organizations due to motivated by following benefits in table 2.4.

Table 2. 4 CD Benefits

CD Benefit	Description
Accelerated time to market	Automation does fast releases, and enables quick customer feedback
Building the right product	The best way of validating requirement is end user acceptance test, and due to fast release the product is as expected.
Improved productivity and efficiency	With automation there is no time waste, and improved efficiency, and team can do more work due to saving time and overall productivity is high.
Reliable release	Pre tested automation is more reliable than manual deployment
Improve product quality	Automation does execute test cases on build time, delivery time and after deployment, which makes high quality product
Improved customer satisfaction	CD has improved product quality, customer gets updates more frequently, customer and team collaboration is improved, and visibility of development progress is high due to frequent delivery, therefor CD gains customer satisfaction by making them aware what's happening on their investment

2.5 Continuous Deployment

The concept of Continuous Deployment has also described by Humble [5]. As per figure 2.6, the only difference between CD and continuous deployment is, CD does production release manually where in here production release is automated. As in [13] the CD means every change committed is ensuring production ready and continuous deployment applies them in production automatically. As in [7], [12], [14] and [23], nowadays many researches used continuous deployment automation approach to efficient their work.

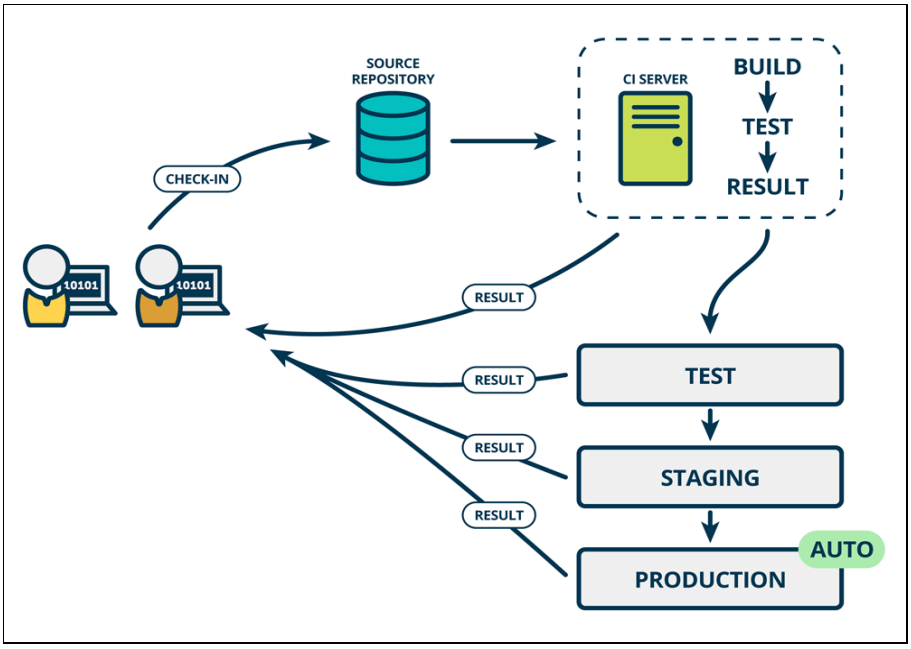


Figure 2. 6 The Continuous Delivery [15]

The continuous deployment has inherited with all the benefits discussed in CI as well as in CD. It enables faster releases where it possible to deploy 1000s of software updates per day [14]. The approach of Facebook in [24] has targeted on rapid deployment of small changes in quickly which helps to identify solution to emerging problems and improve the quality. Rahman et al [16] points that continuous deployment has speed up the processes in agile methods, and mentioning Facebook, GitHub, Netflix and Rally Soft as organizations which are using continuous deployment more efficiently on their production deployments. It is clear that Continuous deployment has more benefits, and it completes the CICD pipeline approach.

2.6 DEVOPS

With the practice of CICD in organizations there is a requirement of having individuals who are capable to perform complete pipeline in CICD, and the new role named “Devops” came into software industry to fulfill that requirement. What is

devops? The term DevOps is the blend of Development and Operations. As per Michael [17], this term did not come about overnight, and it was influenced by Patrick Debois 2009 devopsday conference in Belgium as well as many activities happened during same time period. Suzie P. [15] has described how traditional release process happened, and how it evolved to new way. Figure 2.7, in traditional old way, first developed code is build and handover to QA team to test, when QA team sign off the development it handovers to release team where all the artifacts are prepared and handoff them to operational team. Operational team deploys the artifacts and configurations into production.

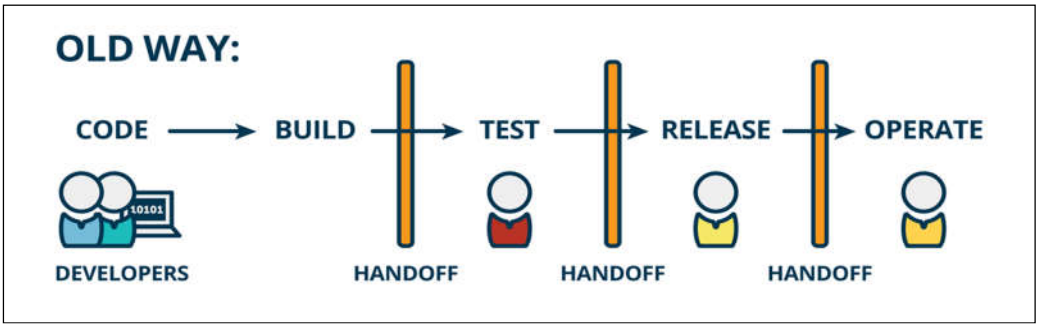


Figure 2. 7 The Traditional way of release process [15]

Although each role has allocated their separate tasks, they are inter-dependent in agile and CICD process. Developer must responsible until development push to production, and each stage developer has to be involved, and QA team also has responsibility of test released item in production, and release team must join with operational team, as they may not aware on new configuration changes and impact. Even after a release, there can be bugs which have to fix, and this same process will iterate again. As in old way, this is not a onetime task, it iterates multiple times as in figure 2.8. In new way the role of devops is more important, and their role is collaboration of development, testing and operational. As mentioned in CD benefits in section 2.4, Chen [6] also described that organization which has adopted role of devops has gain all those benefits more efficiently.

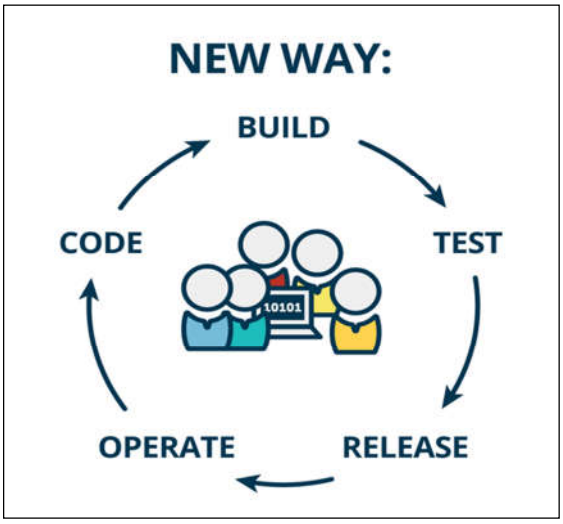


Figure 2. 8 The New way of release process [15]

The role of devops has not limited to release process and it extends with increase of cloud technology usage and new technology adoptions. Richard S. [18] mentioned devops role involves technologies such as, Collaboration, Planning, Issue tracking, Monitoring, Configuration management, Source control, development environment build, Continuous integration, and Deployment technologies. Therefor nowadays the role of devops has been given more importance and highly adopted in organizations which follow approaches such as agile, CD, Data operations, site reliability engineering and system administration.

2.7 CICD Tools

CICD tools can be basically generalized into CI tools and CD tools, but due to CI dependency on CD, it can be classified as Repository and Version Control tools, Build tools, Automation tools, Test Automation tools, and Monitoring tools. Adopting of CICD tools has increasingly motivated with the benefits given by CICD pipeline approach and cloud service support as well as distributed nature has given added advantage to software scale, quality and productivity. Following subsections are analyzed those tools and explains motivation for some selected tools which are to be used in this research work.

2.7.1 Repository and Version Controlling

As mentions in section 2.3.1, maintaining the single source repository is the first practice in CI. To achieve it, there is a necessity of having proper version controlling system (VCS). There are open source tools as well as propriety tools, and those VCS tools can generally categorize based on their repository model and concurrency handle method. Client-server and Distribute are the main two repository models while merge and lock are the main two concurrency handle methods. In open source tools Concurrent Versions System (CVS), Subversion and Vesta are examples for client-server, and Git, BitKeeper, Codeville are few examples for distributed, while Autodesk vault, AccuRev, Panvalet are few propriety tools.

Due to the support of distributed architecture and being open-source software, Git has become more popular and widely use version controlling system with CICD approaches [10], [13], [16], [20-22], [24], [32-34]. Adopting CI practice by using GitLab has benefit of rationalizing infrastructure allocation and administrative operations, improving software development workflow, accelerating innovation cycle and improving the confidence in new software deployment [32]. The approach in [33] has used Git for a CI system while [33] used Git for code checkout when system is migrating to a cloud native architecture as micro services.

Everyone Commits to the Mainline Every Day, is another CI principle which discussed earlier, and it explains the need of having proper branching mechanism to avoid conflicts, improve quality of code and speed up the process while accepting interruptions. The figure 2.9 shows a good example of handling branches in a pipeline which described in [12]. When following agile, the feature wise delivery of software speed up the process while fully utilize developers and testers to improve productivity. Therefore, having feature wise branches are reduced complexity and conflicts. These feature branches should be created from develop branch and once the development complete, it should be merged back to develop. The develop branch maintain the mainline and it merges features to release branch. Finally, once the release is done, release branch is merging to master. In the figure 2.9 f1, f2, f3 and f4

are feature branches created from develop branch and they merge back to develop at D1, D2 and D3 time. T1 and T2 are test environments where T1 does the testing of feature f1, f2 while T2 does feature f3 using develop branch which already has f1, f2. The production environment is P1 and QA phase is denoted by Q1 of release 1.10. The continuation of this feature branch driven pipeline mechanism shows the coexistence of the CICD pipeline.

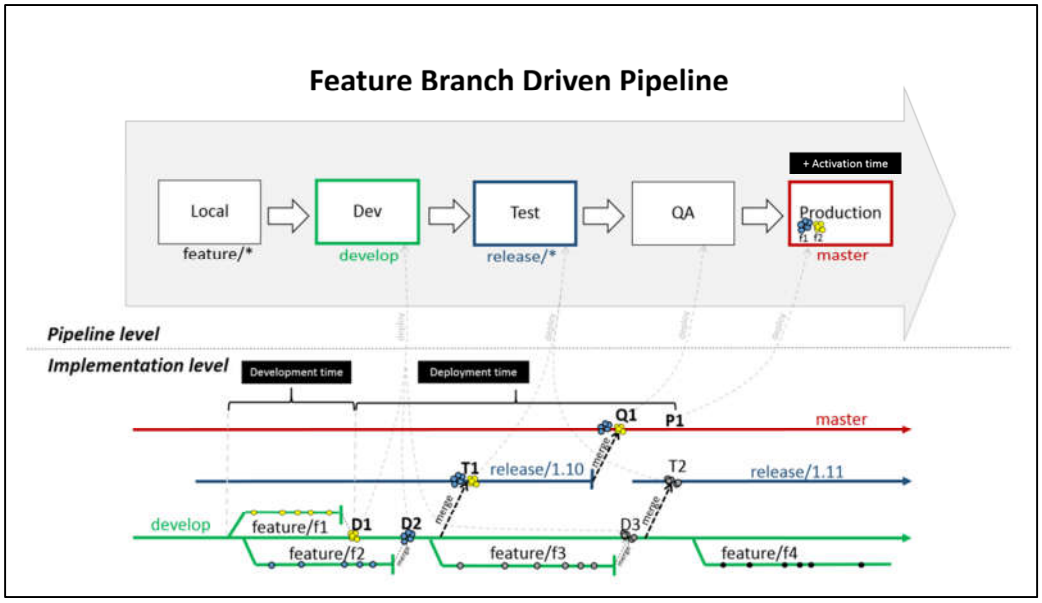


Figure 2. 9 The Feature Branch Driven Pipeline [12]

Having VCS is not enough in CICD, since most of software systems are using third party libraries which need to resolve their dependences at build time. Other than third party libraries, sometimes its mandatory to keep version history of release executable files in case of revert a release or finding unusual system bugs. In this case repositories are used to store those executable as well as third-party libraries where it properly tracks and resolves dependences and keep them up to date. Sonatype Nexus and JFrog Artifactory are the most popular repositories available under OSS. But due to maturity, easy installation, easy configuration in Nexus is motivated to use in research works [22], [35-36]. The example of using Nexus on CD approach is given [35] while [36] use it in a CI related automation approach.

2.7.2 Build Tools

Build tools are the key components in CICD pipeline approach, as it enables the integration and speed up the process. In here, build tool means a build automation scripter or a framework which used in CI process included in CI server. The most popular older build tool is Ant which is integrated with several IDEs. As in [37], Apache Ant is a Java library as well as a command line tool which does the process and steps described in build files, and delivers to target end points. It use in Java applications to compile, assemble, test, execute Java applications as well non Java related programs like C or C++, and in general it is capable of pilot any type of process described in ant script. As an example, Ant is used in [9] to execute test automation of integrating testing in web methods. As mentioned in [37] under the projects section, Ant is used in many forms in several projects such as Ant Commander, Anteloper, Ant-Installer …etc.

Maven is a software project management and comprehension tool which based on concept of POM (Project Object Model) [38]. It makes build process easy, provides uniform build system, quality project information, guidelines for best development practices and allows transparent migration to new features. Maven has integrated with IDEs such as Eclipse, Netbeans, IntelliJIDEA, JBuilder, JDeveloper and myEclipse. Maven is widely used in CI projects since it is more efficient on build project due to capability of speed up the build process by using multi-module maven project like in [39] and it provides more control over dependencies and developers has less effort on build automation.

Table 2. 5 Ant vs. Maven

Ant	Maven
Imperative	Declarative
Script tells the ant tool what to do	What would like to have as the result
Ex: compile the given files and copy them to target folder then make an archive	Ex: mentioned what are the dependent libraries and request to generate web archive
Script base dependency handling	POM based dependency handle

Require to mention library location and specific version, and if need additional reference libraries that has to specifically mentioned	Automatically dependency handle when mentioned initial required library, then reference libraries fetched by it at build time.
More flexible, but it reinvent the wheel	Following POM structure and its own mechanisms may feel too constraining, but it requires less configuration effort
For long term project when scripts become more complicated	Ideal for long term projects as it manage it self
Scripting has no phase, but can make it as phase level execution with complicated scripts	Has its lifecycle and phase level execution, Ex launch integration test on integration-test phase
Require more effort (manual scripting)	Less effort since POM handles everything, and writing a POM is easy

As per table 2.5, we can list down main differences between Ant and Maven, and by considering concepts in each of them Gradle was introduced. Gradle is groovy based domain specific language instead of XML for declaring the configuration. It use directed acyclic graphs to determine the execution order of tasks. It support large multi-project builds and support incremental build. It has more flexibility than Maven, and Google chose Gradle as the official build tool for Android since it has modeled in a way that is extensible in most fundamental ways [40]. Maven and Gradle both are use parallel project building and parallel dependency resolution, but Gradle claims to be performing much better due to following features:

- Incrementality – Gradle tries to avoid repetition of same work by tracking input and output of tasks and does execution in only necessary part. Example: when in multi-model project, when a class of one module is changed, then only that module will build and add into previous archive.
- Build Cache – As mentioned above, it keeps the previous build for reuse purpose, when build the project, it build the changed files only since it keeps the cache of early build class files in cache.
- Gradle Daemon – This is background process in Gradle, it keeps build information available always by keep them in memory.

CI Servers

There is a need of centralized control of all CI and CD phases, actions to avoid unnecessary complication in process. If each tool executes individually, then there can be access permission issue which should provide one by one, as well as manually handling several processes is inefficient. CI servers were introduced to overcome those problems and to have centralized control with automation enabled. Bamboo, Jenkins, Cruise Control, Hudson, Sysphus, Hydra, TeamCity and Tinderbox are some CI servers mentioned in [10]. But in CICD approaches like [11], [13], [16] and [22] has used Jenkins as their main CI server. Jenkins is a self-contained open source automation server which uses to automate all sorts of software related tasks like building, testing and deployment [41]. It is capable of running build tools that discussed like, Ant, Maven and Gradle as well as it executes any kind of scripts which runs through plugins. Jenkins was forked from Hudson which also has similar capabilities and due to following reasons in table 2.6 Jenkins has become more popular in the most of CI approaches.

Table 2. 6 The Jenkins Features

Feature	**Description**
Free open source	Jenkins is free and open source CI tool which hosted on GitHub
Customizable	There is no restriction on how build or process should happen or which environment it should build, and any tasks can be done by integrating plugins which require running the process.
High community support	As it is open source and been in the field many years, it become mature on process and developments while adding more features and supported by many communities.
High Resource Availability	Many communities have discussed processes and bug fixes, and there are lot of tutorials available in online
Support VCS	It supports most popular version control systems such as SVN, Mercurial and Git.
Scalable	Jenkins has master slave capability where it can execute distributed builds more efficiently.
User controlled chain of automation	Jenkins allows developers to describe their chain of automation in text form, which promotes automation

Easy installation	It requires Java runtime and Jenkins.war file which can install within minutes.
Plugins support	Jenkins supports vast number of plugins which helps to run many types of tasks and processes.
Cross-platform builds support	It is a Java application which can run in any Operating system or platform where JVM exists.
Supports technological stacks for free	Jenkins support many technologies like Docker, Amazon EC2, and S3 by using specific plugins.
Security	It is host who controls the Jenkins, therefore self-hosting provides safety to store key environment variables, and has credentials and plugin which make it secure.
Support Multiple test runtimes	It allows adding several runtimes at global configuration options where it can use in different environments by specifying in related job configurations.

2.7.3 Automation (Configuration Management)

Automation can be considered as part of a build tools since most of CI servers are created to make process automate. The jobs in CI servers are the trigger points of automation scripts and server control the process like initiation, termination and pause. When VCS maintain the code and repositories keeps executable files and when build tools create artifacts and when CI servers can able to integrate all the features and tools together and initiate automation, what are the things remaining in deployment automation? If we consider a large scaled system of having thousands of servers which needs to be update during the release process, is it possible to do it within a day in manual work or even automate scripts. There are so many configurations which need to be change during a single release. Example, suppose that a simple release require to add new data source to in HA proxy in each server where having 100's of application servers. This simple task may take more than half a day if it's going to do in manual. It may takes only few minutes when using scripts, but if there is properly configured and well managed tool used, this simple task can

be done within few seconds. Therefor what will be discussed under automations is all about configuration management (CM).

What are the configuration management tools? It is a tool which enables to manage infrastructure deployments through their design, implementation, testing, building, release and maintenance. As explained earlier, after a cloud environment setup is done, developed software system has to be deployed. But before that some configurations has to be done on server like proxy configurations, load balance configuration, data base connectivity and pooling, 3rd party service connectivity, host management, link redirections likewise there are list of items need to be configured before software deployment. Doing all these tasks by hand is time consuming and introduces unnecessary risk and complexity as well as it is expensive. Therefor main objectives of having CM tools are cost reduction, reduce the complexity of release process, proper host management, and easy configuration, reliability, save overall deployment time which improves the efficiency of process, and optimize resource utilization which improves the productivity of an organization.

Google has given a solution for manage compute engine in their cloud platform in [43] using CM tools puppet, chef, salt and ansible. The compute engine is used to create virtual instance on their cloud platform when provide necessary details for server creation. There are main two models suggested named master-agent model and standalone model. As in figure 2.10 the master-agent model has a master server and it connects to a database which keeps track of all the information of CM process status, and master server provides required configurations to all client instances. This model has more transparency since administrator can perform installations, configurations, process status tracking by connecting to master, and master connects to compute engine which used to create required virtual machine. In general practice, above compute engine can be replaced by a server provisioning system with CM tools.

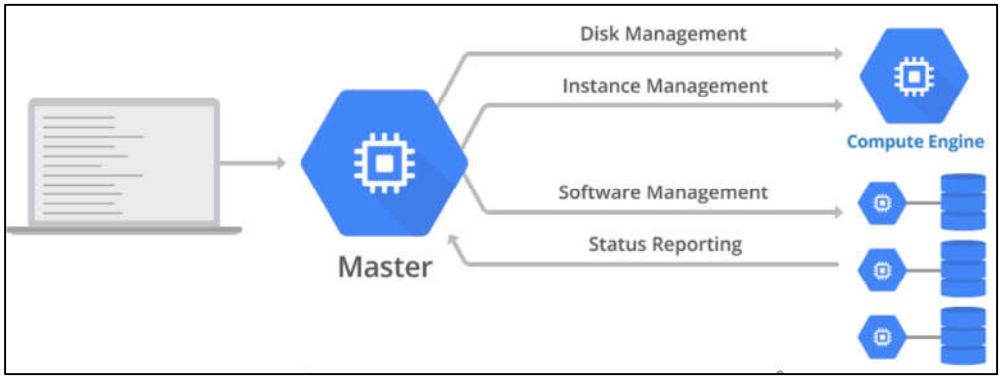


Figure 2. 10 The master-agent management of Compute Engine [43]

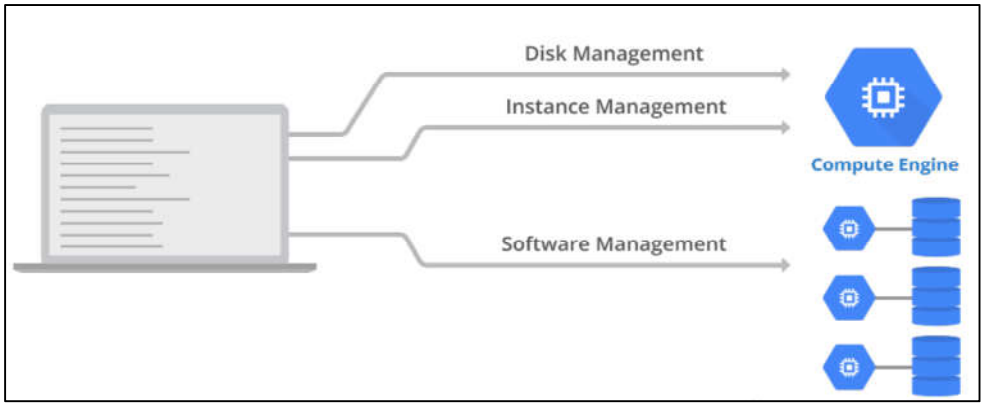


Figure 2. 11 The Standalone Management of Compute Engine

Also this master-agent model can be implemented as pull or push model. In pull model there is an agent which installed in client and it periodically check for new configurations. In here, when client agent notices that there is an update, then only configuration will retrieve and apply on client instance. But in push method master has ability to push changes directly to clients using a middleware or message queue and changes are expected to be applied sooner than pull method. As in figure 2.11, there is no master server in standalone model and client server may connect to one or several workstations in the environment. Though standalone model does push operations it can configure in such a way that clients use it to pull the updates from workstation with CM tools. In here, the puppet and chef are pull types and salt and ansible are ‘push’ method CM tools. CM has several features as in below table 2.x

Table 2. 7 CM Features

Feature	Description
Enforcement	It avoids configuration drift, i.e. having same consistent changes in unique cluster, it reduce surprises of having inconsistent issues.
Enables cooperation	As failures reduced and there is no time waste, operation team and developers cooperate when in releases.
Version Control friendly	Version control is best practice with CM, when revert a release or when in issue tracking, versions makes efficient the process.
Enable change control process	When it VCS friendly, it allows to pre-review the infrastructure changes and control the unnecessary changes on production
Abstraction	It can do necessary changes on different operating systems and environment while abstracting details of CM host system

CM Tools

- CFEngine

As in [44], CFEngine was the first modern open source CM tool released in 1993. It provides automated configurations and maintenance of large scale systems like consumer and industrial devices, embedded network devices, smartphones and tablets. CFEngine has a capability of make the infrastructure or resources into desired state of designed, by simulating the configuration changes before actually apply, providing automatic self-healing capability while continuously correcting configuration drift, and collecting reports on difference between actual states and desired states. CFEngine was developed with C and due to that reason it has smaller memory footprint and it runs faster and also has less dependencies.

- Puppet

In 2005, Puppet Labs has released Puppet as software and resource management tool, and currently its core functionalities are free and open source while providing additional tools and features like GUI, API and command line tools to manage nodes on Enterprise version. Puppet is a declarative language tool which designed to manage configurations of Unix-like and Microsoft systems. Puppet use a manifests

to store descriptions of system resources and states, and a utility called Facter has used to discover system information. It compiles manifests into a system specific catalog where it applied containing resources on target systems. Puppet has client-server architecture where it can behave as master-agent model, and it also can use as standalone application. In [43], Google has used puppet in both master-agent and standalone models.

Since puppet is being an open source and it is flexible, there are many custom libraries and modules available. As it tracks each process steps in nodes like provisioning, installations, up-gradation, and maintenance, it increases the manageability. It improves the efficiency by automating repetitive tasks and increases the productivity by reuse resources across different platforms. Puppet can be found in many research work related to CI and CD, in [10] puppet is listed as one of configuration management and provisioning tool, in [13] AWS has mentioned puppet as one of the CM tool which should skilled by development teams in automation. In [22], puppet is classified as Infrastructure automation and CM tool, while in [39] mentioned puppet as orchestration tool. When choosing a CM tool for control system approach in [44] puppet is tested parallel against other OSS and in [46], puppet, chef and Ansible features has been compared.

Puppet can be consider as a choice when system needs stability and maturity, and its ideal for large scaled heterogeneous environment, but it needs more skilled team. As pros on puppet, well established support community, has maturity and runs on most of operating system, simple installation, having complete web UI and has reporting capabilities. When using puppet we need to understand ruby when need to use CLI for advance tasks, DSL and complex design makes hard to understand when increase the scale, and model-driven approach is less control over code-driven approach, are considered as cons of puppet.

- Chef

Chef is open source software and resource management tool which released in 2009 with capability of running in Unix-base as well as windows platforms. It has written in Ruby and Erlang, and used domain specific language (DSL) for writing

configurations. It integrates with cloud based platforms such as Internap, AmazonEC2, Google Cloud Platform, OpenStack, SoftLayer, Microsoft Azure and Rackspace to server provisioning and configurations. A “recipes” is a simple configuration file which describes ‘manage server’ applications and utilities. By grouping several recipes a cookbook will create for tasks like package installations, resource management, and correct the configuration drift. Chef can run as either client-server mode or standalone, but Chef is more popular in pull method. In [10], Chef is listed as one of configuration management and provisioning, and AWS in [13] has mentioned chef as one of CM tool which should be skilled by development teams in automation. In [39] mentioned Chef has more abilities and features in paid version than free version, and was pointed as one of the drawback on Chef over Docker, and as in puppet description Chef also is tested parallel against other OSS in [44].

Chef is good mature solution for heterogeneous environment with development focused teams. As pros of Chef, it provides rich collection of modules and configurations, its code-driven approach gives more control and flexibility on configuration. It has strong version control capability as it binds to Git, and provides easy installations through knife tool which works with SHH. As cons of using Chef, though it gives a benefit of version control with Git, it makes huge dependency for using Chef, and it has high learning curve as team needs to familiar with ruby for configurations. Therefor it’s not a simple tool, it can leads to complex large code base with increase of scale and heterogeneous environment and as it does not support push functionality can expect minor delay on configuration application on all servers.

- SaltStack

Salt is python-based open source CM tool and remote execution engine which released in 2011. It used machine readable definition files or Infrastructure as Code rather than physical hardware configuration or software configuration platform for managing and provisioning of servers. Salt also can setup as master-client model or standalone model, and it uses push method and SSH to communicate clients. Salt has a capability of applying configuration on group of servers which is more efficient on

deployments. Therefor salt can be ideal solution for large scaled system configuration. AWS [13] has mentioned Salt as one of CM tool which should be skilled by development teams in automation and in [44] Salt is compared over Ansible. As pros of using Salt, less rework, Once the Salt setup is done usage is straight forward, configurations are consistent since all are in YAML, has good introspection, has strong community and high scalable. As cons, initial setup is harder than usage, and documentation are complex, web UI has only simple features, and it is not support all non-linux operating systems.

- Ansible

Ansible is a software automation tool which released in 2012 and has capability of automating software provisioning, configuration management and application deployment. As most of CM tools Ansible also has two types of servers, which are controlling servers and nodes. Orchestration starts from controller machine and it manages nodes over SSH, and details of hosts and nodes are mentioned in files called inventory. In orchestration, Ansible deploys modules to nodes which store them temporarily while communicate with control machine through JSON protocol over standard output. Unlike other CM tools such as CFEngine, Puppet and Chef, the Ansible use an agentless architecture. Due to this reason, when Ansible is not controlling node, it does not consume any resource of node since there is no agent, daemon or any other background program executing for Ansible. Therefor it reduces the network overhead by preventing nodes polling from control machine.

Ansible has designed with goals of minimal in nature as it removes additional dependencies in management systems, it is secure due to there is no agents and only SSH makes communication, it is simple since only python requires on nodes, it is highly reliable as ansible playbooks ensure unexpected side effects on nodes, it has low learning curve due to YAML based. Ansible is used in several research works as their automation approach. As in puppet, chef and salt, AWS [13] has mentioned Ansible as one of CM tool which should be skilled by development teams in automation. When evaluating low learning curved python based tools in [44], ansible is selected over Salt due to feature of quick start and simplicity, and [45] has used

ansible for automation of a bioinformatics framework which deploy on cloud. In [46] ansible is selected over puppet and chef as it is the available simplest solution for the configuration management.

Ansible is the ideal solution when we need to setup and run something quickly without having agents in nodes. When an organization needs to execute over vast number of servers streamlined and fast, then Ansible will be the best solution. It is agentless and SSH based easy learning curve due to usage of YAML, structure of playbook is simple and clear, it can handle more repetitive tasks which allows teams to focus on other tasks, variable registration feature enables tasks to register variables for later tasks, and has more streamlined code, are the pros of using Ansible. When consider the Cons of Ansible, since Ansible does not depend on tool, it makes less powerful and has performance speed concerns which cannot control externally and has no consistency on formats of input, output and configuration files. It also use DSL for logics, which mean, new members has to refer documentation frequently. Since variable registration is mandatory for every function, it makes complicated for simple tasks, and its introspection is poor as it is difficult to see values of variables within the playbook.

- Docker

Unlike other CM tools Docker is a container which has capability of creates and execute of container for any software solution. It has additional layer of abstraction and operating system level virtualization with resource isolation feature which allows independent containers to execute avoiding the overhead of starting and maintaining of virtual machines. Docker was launched in 2013 and due to the reason of lightweight containerization technology it was vastly adopted by software industry. It is scalable, all the configurations were in container and its easy on deployment by creates image copy and distribute, and less deployment effort are few benefits. Docker also has used in several research works, in [22] docker is mentioned as CM tool which can be used in CD process while in [23] docker is used with swam orchestration, and in [39] docker is used to build a CI and deployment pipeline.

- PowerShell DSC

The windows PowerShell DSC (Desired State Configuration) is Microsoft extension which released with Windows Server 2012 R2 and Windows 8.1. As it is propriety, it has less community support, but Microsoft has given more options to extend DSC features by exporting declarative configuration scripts in a form of puppet or chef, due to that reason PowerShell also has become a part of CM tool discussions. DSC enables to install or remove server roles and features, and able to manage registry settings, files, directories, processes, services, local groups, user accounts, packages and environment variables. It also has capability of execution of PowerShell scripts, and it fixes the configurations which drifted from desired state by discovering the actual configuration states. DSC allows both pull and push methods, and it use Local Configuration Manager in client node to connect through IIS server and retrieve the configuration desired states, and in push method it may works as agent less.

Though we have many options of CM tools, but selecting CM tool for application is most difficult task. Many research works has compared these tools like in [39], [44] and [46]. Most of times it's compared between puppet and chef [39] or puppet, chef, Ansible [46], and in [44] has compared puppet, chef, Ansible and salt, and that means understanding the pros and cons of puppet, chef, Ansible and salt has an additional advantage in terms of scalability, ease of setup, availability, management and interoperability. The below table 2.4 has compared those feature in puppet, chef, salt and Ansible, and all tool has similar in terms of scalability, Availablity, Management and interoperability, except Ansible is fast and easy on setup. Due to that many communities, many research works and most of cloud providers has adopted ansible, and in this research work also has selected ansible as main CM tool.

Table 2. 8 The CM tools Comparison

Scalability	Puppet	Highly scalable, multi master, multi agents
	Chef	Highly scalable, multi chef servers, multi clients
	Salt	Highly scalable, multi master, multi minions
	Ansible	Highly scalable, multi instances, multi nodes
Ease of Setup	Puppet	Hard, Install puppet server, agents in each client
	Chef	Hard, Install chef server and client, extra client for workstation for prerelease test
	Salt	Hard, Install Salt master, salt minions
	Ansible	Easy, Only Masters or instances, no agents, fast setup
Availability	Puppet	High Available, Multi master will replace master in failure
	Chef	High Available, Multi Chef master server replace in failure
	Salt	High Available, Multi master configuration replace master
	Ansible	High Available, When Primary down, secondary instance replaced
Management	Puppet	Hard, Pull configuration, own DSL
	Chef	Hard, Pull configuration, Ruby based DSL
	Salt	Easy, Push configuration, YAML based DSL
	Ansible	Easy, Push configuration, YAML based DSL
Interoperability	Puppet	Exist, Master in linux only, but agents can on windows
	Chef	Exist, Master in linux only, but clients can on windows
	Salt	Exist, Master in linux only, but minions can on windows
	Ansible	Exist, Primary server in linux only, but nodes can on windows

2.7.4 Test Automation

When organizations try to achieve faster time-to-market in agile methods with CICD, the automation of test cases has provided huge efficiency in the process. The idea of "test automation pyramid" in agile process was introduced by Mike Cohn [25]. This pyramid consists of three different levels as Unit, Service and UI. The Unit tests are at the bottom of pyramid where can have many unit test cases per single development cycle, and Unit test writing is very easy as well as its less cost. Middle layer is Service where system tries to integrate with other services like third-party services, data services. The test automation of the service layer is difficult task than Unit tests and its takes time. The top layer is the UI, which is the costliest and can have very few test cases, since UI test is brittle, expensive to write and time consuming.

The extended version of test automation pyramid in figure 2.12 suggested by AWS [13] has introduced another layer called Performance/ Compliance on top of service test layer. To have complete CICD pipeline, having a performance test phase is very crucial requirement. The test automation can perform in build time, staging time, and production time. At build time unit tests and analysis test can be performed while in staging time can perform testing such as Integration, Component, System, Performance, Compliance and User Acceptance test. In the production final Canary test can be performed on new deployed servers. The table 2.5 shows some tools which can be used in each test levels.

Table 2. 9 Test Automation Tools

Test Level	Tools
UI	Selenuim, Autoit, SAOtest, Sahi
Performance	Jmeter, LoadRunner, LoadUI, LoadComplete, OpenSTA
Service	Testingwhiz, SoapUI, postman, httpmaster, vRest, storm
Unit	JUnit, TestNG, JTest, JWalk

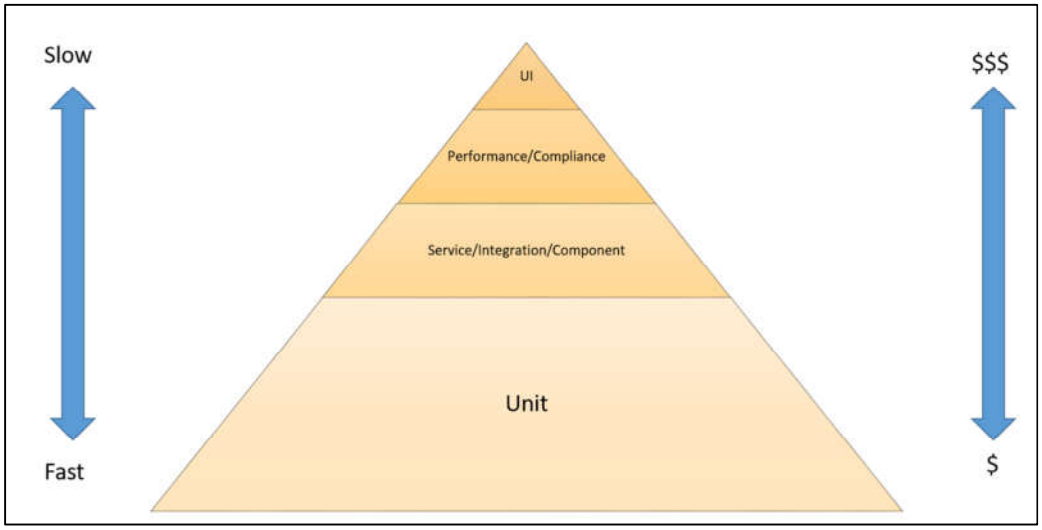


Figure 2. 12 The CICD Test Pyramid [13]

The test phases and levels can be varying with the application and the technologies that used. Test automation orchestration approach in [26] has described deployment pipeline with Unit test, functional test, Sniff test and Performance test phases, and also it is well scaled to perform any number of test types by using available resources optimally. The continuous testing approach described in [7] has benefit of stable code base, faster response and easy decision making, and both [7] and [9] has suggested TDD as development strategy to use with CI to gain the quality of software.

2.7.5 Monitoring

In CICD approach, the monitoring has an important aspect since it can be used with automation in system installation and configuration, as well as it provides many benefits. In [27] has mentioned cloud providers and application developers has gained following benefits when using system monitoring tools. It helps to keep organization infrastructure and applications operating at maximum level of efficiency. It detects variations in behavior of infrastructure and application

behaviors which can impact on performance. It tracks the services and infrastructure failures and new additions, and other dynamic configuration changes and accounts the Service Level Agreement (SLA) since it can identify violations in various QoS parameters.

Aceto at el [28] explains, cloud systems need monitoring since it helps on capacity and resource planning and management, data center management, SLA management, billing, troubleshooting, performance management and security management, and also it needs cloud monitoring since it has nonfunctional features such as scalability, elasticity, adoptability, timeliness, autonomic, comprehensiveness, extensibility, intrusiveness, resilience, reliability, availability and accuracy. Other than those benefits and features, the fast responsiveness of monitoring tools through emails, SMS has elaborated in [29] by using a request tracker and Nagios monitoring tool.

There are various commercial available monitoring tools such as Monitis, Reveal Cloud, Cloud Harmony, Nagios, Nimsoft, Cloud watch, Open Nebula, Logic Monitor and many more mentioned in [27], and those tools were evaluated against dimensions like network architecture (centralized/decentralized), interoperability with multi clouds, visibility in multiple layers, support SNMP and having extendable API's. Based on that evaluation Nimsoft and Nagios have supported all the given dimensions, and among both only Nagios is an open-source tool. Nagios has vast community support as it is an open source and it was used in many research works [16], [27-31]. Mikko [30] has used Nagios to evaluate a self-healing system by tracking and monitoring of faults and checked the automate system recovery options, while in [31] has used nagios for intrusions detections. In [16] which is CICD related research work also has considered Nagios as the monitoring tool.

CHAPTER 3

METHODOLOGY

This section describes the approach of achieving the defined objectives under research objectives section. As described in literature, the CICD pipeline approach has been used in many research works in last few years due to its wide area of applicability with different approaches. Many software organizations and cloud providers were motivated to adopt CICD due to its huge benefits which gains through it. CICD has defined a new process standards and practices which are customizable, efficient and productive in nature which leads to extend its features to solve various problems. Therefore, this section will describe the methodology of how CICD approach is used to solve problem of automatic horizontally scaling of services in agile Sprint deployment, under a system which performance is mainly concerned.

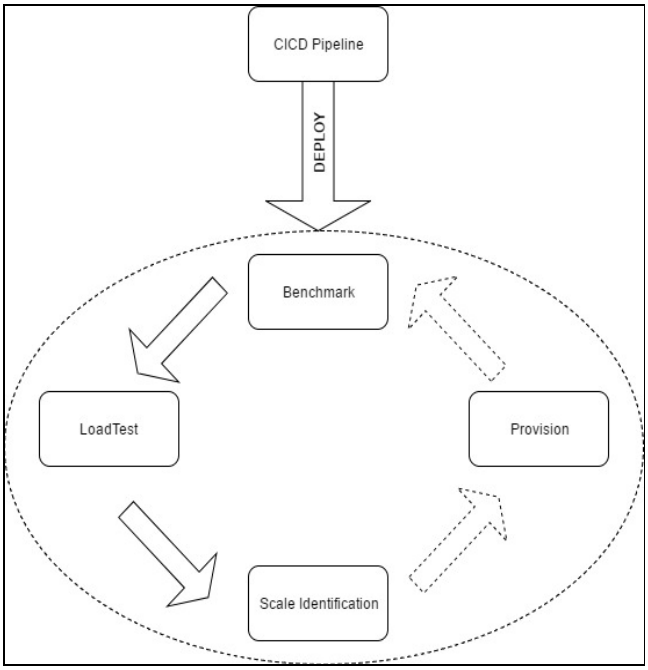


Figure 3. 1 The Deployment scale with CICD

In this approach CICD is used as main deployment process pipeline methodology. CICD used to deliver developments to various environments, and at the production release it can be deployed in one of the methodologies described in section 3.1. When deploying a system where performance is concerned, the deployment should always consider a rolling of two levels. First level is that deploying to test-bench environment and evaluates the new development, and then second level is for completing production deployment. The main benefits of deploying in two levels are early identification of performance issues in latest sprint development and then fix those issues. Sometime fixes that can be applied on development may become a separate performance optimization task and sometimes existing system already in optimized level as per team capabilities. In those cases, team may try to evaluate the option of scaling. But how to scale, in what extend it should be scaled, what is the current production performance status, are few questions among the hundreds of them need to be answered before planning the deployment scale.

The figure 3.1 shows the deployment scaling solution with CICD which is proposed by this research work. According to it, benchmark, load-test, scale and provisioning are four iterative phases which should follow with deployment scaling. In the benchmark phase, it checks the production capability and limitation, and then identifies what is the current benchmark level of production, and that level is used when define the scaling requirement. The new software will proceed through load test phase, to check whether new software can be performed same as production or to see any performance issue. The scaling factor is used to check whether new software requires additional nodes or not, by using the bench mark levels and the load test result took at early phases (it does not scale, but it identifies the requirement of scale), if it requires new servers, and then it will provision them in next phase. The outer boundary circle is for deployment and it is denoted in dotted lines as there can be several deployment methods which can be applied on this process. Since provisioning of new servers are to be done only when it identified a need of new servers at scaling phase, the arrows went through provision state are also showed in dotted lines. Those phases are to be iterative as it should be followed in each agile deployment. When the system has trend of having peak seasons, this iterative

approach is added advantage to perform an elastic scaling based on system load demand, which is more cost effective in cloud systems. Introducing of new level before production release may consider as additional effort, overhead and time consuming. But having proper automations in CM tools with CICD pipeline can save the time and it reduces the risk of deployment failure and make sure having stable production release.

3.1 Deployment methods

At the end of each sprint, all teams are busy on release preparation tasks such as collect configurations, collect database changes, code merging, build verifications and waiting for client approvals. All these preparations are for achieve one goal, which is, a smooth release or a release with minimum impact to existing production. How to do a deployment with minimum system impact, or could it be done without a system downtime. That decision must make at the sprint slicing level, to identify which approach that suitable for current sprint deployment. The application or the development method has defined the approach of deployment, and in most of situation it could be a custom deployment strategy which made up with collaboration of several strategies. Ville [19] and AWS [13] have mentioned several deployment strategies which are using in Continuous Deployment as in table 3.1.

Table 3. 1 Deployment Strategies

Deployment Method	Description
Feature flags	Use a flag with enable/disable on new feature to control the impact on production. Flag can store in database column with true/false statement or use a configuration files like property files. Introduce new API method for new feature also a common feature flag practice.
Dark Launches	Provide limited group access on new feature to identify risk on new development at production.
All at once	Replace all services with new version, and requires down time though it cost effective with automation.
In place - Doubling	Having mirror image of service clusters and run both old and new versions parallel. After new feature tested on production

	decommission old cluster later. AWS recommend this method.
Rolling : In-Place Rolling	Deploy new version to few servers and test it, once testing completes deploy to full system
Rolling : Canary	In [5], it releases first only to a small group of users by automation, and monitor the new release for some time, and if there is no problem with new version, do the complete production release. AWS [13] also defined with same idea by mentioning that, initialize release canary on small number of servers or nodes, and then do the verification, and once the all verification passed, do the gradual release on rest.
Rolling : Rolling with an additional batch	Release new software into newly created identical cluster / server, and then test it by allowing it to live traffic. Once testing completes, do full system deployment except for last cluster / server since initially already added new one. This method is suggested by AWS [13].
Immutable Deployment	This is similar to rolling in-place doubling, except in here it does not allow to run both version parallel. In here, it switches from old version to new through load balancer. This method is recommend in infrastructure upgrade or when moving to different software architecture.
Blue-Green Deployment	Use when system upgrades or when software architecture has been changed. Similar to immutable, old version is Blue, new is Green. Switch the environments using load balancer swap or DNS swap. Though deployment hard, roll back is easy.

3.2 Benchmark

In general, benchmark is the process which compares the one system parameter over another, performed under identical conditions. It shows the relative performance of one system which was accepted as better perform based on business and other technical requirements. It can be considered as a part of test phase in SDLC, and when perform benchmark testing it needs to perform same environmental parameters under same condition to compare results. It can be repeatable, as it is not a onetime task, and it needs to repeat after every release or every hardware software upgrade to identify current performance status of system. It is the main factor which tells,

whether system has gain performance improvement or degrade the performance at each delivery. It should perform on identical infrastructure, since it helps to identify factor of how much time system is performed comparative to existing or earlier results. When performing benchmark, it should be isolated, and it should not be interrupted by other processes which makes incorrect statistics. The database level transactional locks has highly impact on performance, therefor when running a bench mark there should not be having any external interference on benchmark results.

In most of the systems, database related processing are the biggest performance concerned, as example, Cooper et al [49] used yahoo cloud serving benchmark (YCSB) framework to evaluate the most popular new generation cloud data saving systems such as Cassandra, HBase, PNUTs and simple shared MySQL. In [48] mentioned several benchmark methods, performance metrics and relative approaches, which are gathered using systematic literature review method to collect metrics and adopted methods in other research works. It categorized those metrics under performance features for cloud service evaluation, as example communication, computation, and memory and storage were considered as physical properties while transaction speed, availability, and latency, reliability, through put, scalability and variability as capacity properties. Some cloud systems were evaluated using various benchmark systems, example [50] has evaluated five benchmarks on Amazon EC2 namely YCSB, CloudSuite, HiBench, BenchClouds and TPC-W, and it mentioned the fact that scaling up is more cost effective in sustaining form higher workload in cloud servers.

In here, we are trying to find is the limit of current production servers or cut off points. The system is expected to perform with same latency while having availability when increase the load on system using goreplay tool. When increase the load, if a system becomes unavailable or latency dropped from expected level, then that point load has been considered as system benchmark load. In those cases, CPU and memory usage also has to be concerned, as system get unavailable due to one of those has exceeded their limit, and those two parameters behavior can be monitored through nagios monitoring tool.

3.2.1 Duplicate Traffic

In this research work the process of benchmark and load-test are depend on live traffic than simulated traffic. Since the test bench environment is identical to production in infrastructure level as well as in configurations, when compare and evaluate performance of new software, it is necessary to having production traffic to serve both in equal. Therefore duplicate of live traffic is the best option as it does not create any impact to production when compare to method like traffic splitting. There are several open source tools which can be used on this purpose such as tcp-replay, bit-twist and goreplay. Tcp-replay is a tool and a package which contains several other tools like packet capturing and analyzing, and it requires some additional learning to fully execute whole features, Bit-twist also has similar capabilities. But in this research work goreplay has been selected as it is simple to configure and execution can be done by one command line, while it has different traffic duplication methods over http traffic. In shared environment, when an application duplicates traffic, it impacts on system overall performance due to system caching. Since benchmark and load test performed in same environment, the cache hit ratio can assumed as equal in both phases. Therefore, impact of cache on performance analysis when duplicate traffic has to be separately studied, and it is not captured in this approach.

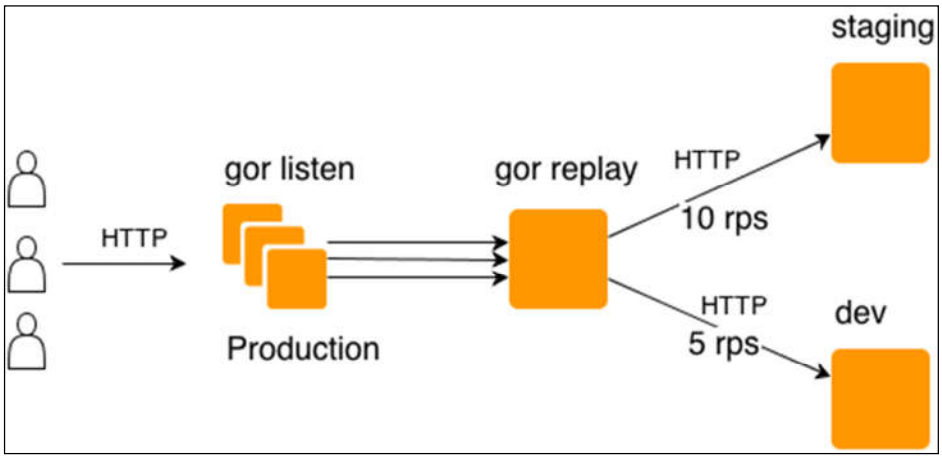


Figure 3. 2 The GoReplay Traffic Duplication [47]

GoReplay [47] is an open-source tool which can use for capture and replay production traffic into test environment in order to perform continuous test the software with real production data. It is the simplest and safest way of test the application using real traffic before release in to production as it does not require additional configurations or coding. When the application scale becomes high, the effort that required to test also high, and goreplay use simple idea of reusing existing traffic for testing. As per figure 3.2 goreplay can listen in one server and replay it to multiple servers with various proportion as we required. It shows that ten request per second replay to staging environment while 5 request per second for dev environment. There are no configuration files or any other configurations, how much proportion and how many environments need to replay is send as parameter of one command execution line, and those commands are straight forward, simple to learn, and it can be used in three different ways as below.

- Capture web traffic

Goreplay can capture the traffic by listing to given port (usually port 8000) and then log it to standard output which can be seen by opening a browser under local host and given port, or can be seen it in terminal by calling curl the URL. To do that, goreplay does not require any additional third party tool.

- Replaying

It can relay the original traffic from one environment to another environment. It can replay from one server node to another server node, by specifying source and target required IP and ports in execution command, and also as mentioned earlier it can replicate portion of traffic or percentage of traffic from environment to another, in that case it is capable of increase the traffic by copying the original traffic.

- Save to file and Replay later

Suppose that system hits the peak traffic at midnight and team needs to having that peak traffic to evaluate some test scenarios. In that case, goreplay can be used to save the traffic at midnight by executing cron job with shell script which included the

goreplay traffic saving command. Then in day time, they can recreate that peak traffic by replaying the saved file. In this case traffic replay in same order with exactly same timing as they were recorded in peak time.

3.3 Load Test

Perform a load test on each software delivery is the best practice for systems which performance is more concerned. Load testing identifies the new software limitations and it benchmarks the existing system. Load testing, stress testing, and spike testing are types of performance testing [13] and they are using for benchmarking under predefined criteria. Load testing is a one of key feature in cloud-gen automated test engine approach in [9] and it used Jmeter for load generation. The load testing is expensive and has limited access [7] as it needs to perform in production identical servers to having more accurate result, but the approach of service virtualization can remove this constrain as it can simulate the environments and increase the efficiency of testing process.

Most of load test approaches have used the simulated load which is highly deviated from production actual load behavior. Sometimes performance issues exist in production may not be able to identify by that simulations as the exact load pattern cannot replicate or due to differences between environments. But still that simulation is worth since it can finds performance issues and system limitations under standard conditions. This approach suggests performing load test using live production traffic with goreplay. As explained earlier, go-replay is capable of duplicate traffic without any impact on host system. Therefore in here, goreplay is used to perform load test with live traffic on production cloned environment with same configurations and specification. In next chapter explains how load test perform with ansible automation in more details.

3.4 Scale Identification

The main purpose of this phase is to identify the requirement of system horizontal scaling, since scale out with over provision gives more user satisfaction as well as its easy to provision since adding identical new server with same configurations is easy, when in scale up, using the same configurations may not logical and it's a resource waste. As in figure 3.3 which described in [50], when system scales up, the over provision which marked with red bars and under provision with blue bars can be happen, and scale out has more tendency of over provisioning but it safer than under provision behavior with scale up, and it suggests a brute force approach with automatic scaling as best resource utilization method. [50-52] approaches are discussing the auto scale methods or elasticity in cloud systems. As per [51] the elasticity is the ability of system automatically provision and de-provision the computing resource on demand. It has evaluated the elasticity on resource demand against scalability and efficiency. The decision of what are the servers to be added and what are the servers to be removed under on demand, has explained in [52] by using Markova Decision process to determine optimal scaling approach.

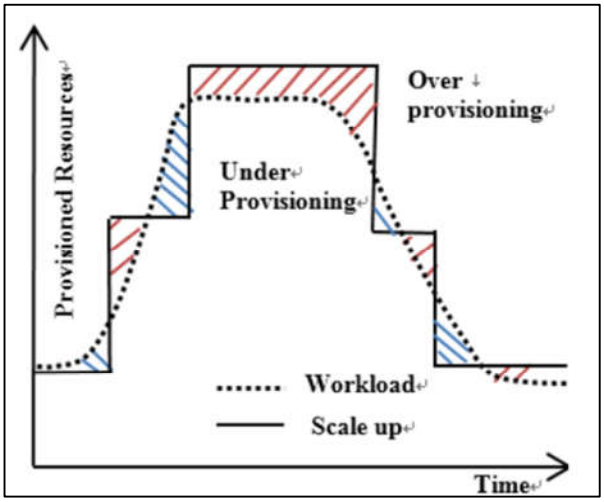


Figure 3. 3 The Over Provision and Under Provision [50]

In this research work use simple proportionate method to identify the requirement of scaling and how much of additional nodes require to perform system at expected level. For that proportionate calculation, the result of bench mark process and the results of the load-test process have been used, as both processes were performed on identical servers under same load and only difference was the application version. The final calculated require number of servers are encouraged to have over provisioned as it satisfy the client than unnecessary performance issues at under provision. Therefore, the results will be rounded to ceil to get next integer value. When new software version perform better than existing version, it calculate the number of nodes that can be removed as negative values, in this case result will be rounded to floor to avoid under provision.

The scaling requirement calculation method:

Current benchmark load per server = Z

The maximum load could handle in load test = Z1

The total load of the system (number of nodes N) = Z x N

Load handling by new version in existing nodes = Z1 x N

Additional nodes requirement Zx = (N (Z – Z1))/ Z1

Since number of servers cannot be a decimal value, Zx should always return next (round to ceil) integer value from the result.

Example:

Z = 10,000 Z1 = 9,200

N = 20 (Four clusters with five nodes each)

Zx = (N (Z – Z1))/ Z1

= (20 (10,000 – 9,200))/ 9,200

= 1.739 => 2

In above case, it requires 2 additional nodes to keep the expected performance with new software version. If result is -1.739, and then it will de-provision only one node to avoid under provision.

When defining the maximum load which can be handled by the system as in above calculation, the latency or the system response time has been used as main performance static parameter. But CPU and the memory usage are the dynamic parameters which impact on system availability as well as on response time. As in below figure 3.4, most of systems get into their saturated states in shorter time period and CPU raised and come to peak level. The memory usage also increase or free memory get reduced and stay in saturated level with minimum free memory. Though it looks unstable of system availability at this stage, this pattern of peak CPU and low memory stage could able to keep system alive since proper process thread handling mechanisms in applications, as well as the continuous system garbage collection which free up memory. But at this stage when increase the load of the system, there can be situations of process waiting which drags the whole process pool keep wait and lead to slower responses, and there is a possibility of system getting unavailable as CPU reach its limit due to process wait. Other than that when there is situation of increase in system disk IO or increase of internal threads can cause the termination of system process due to reaching CPU limits.

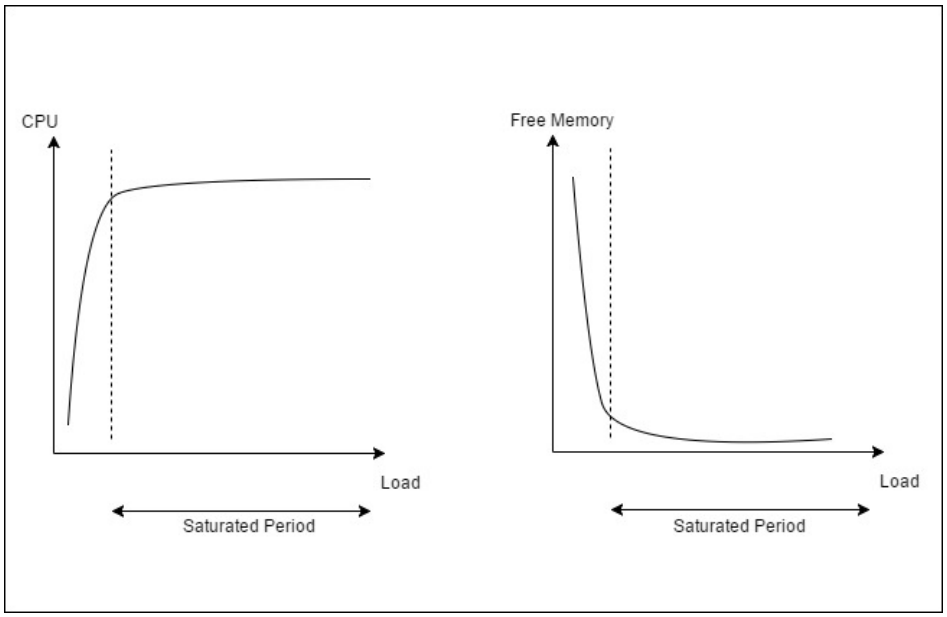


Figure 3. 4 The CPU and Memory behavior in Virtual Environment

When a system uses resources at the maximum level, while maintaining the response time, gauging the scaling level based on load at expected latency is the best method of calculating scaling requirement. Because the deviation of CPU and memory level after it comes to saturated level is difficult as it shows very minor difference with time. Therefore, behavior of CPU and memory cannot be considered for calculation of scale level. But there are some systems which use redundant resources, as example, some systems use only 80% of CPU and memory while keeping rest of resource to face unpredicted loading conditions. Those particular systems often use virtual servers with multi cores CPU and large memory, and when resource utilized more than 80% it initiate to provision new servers with same specs. Since those systems perform better with redundant resource usage, it takes more time to become saturated level with unpredicted loads, and it gives more chance to apply scaling options based on resource utilization level (mainly CPU and memory). Since this research work only focused on systems where performance is concerned in this phase it will only evaluate the scaling options based on response time at given load through automation while opening a new forum for auto scaling options base on CPU and memory utilization.

3.5 Provisioning

Provisioning is the method of providing or allocating resources in virtual environment. In cloud provisioning, the allocation is done by cloud provider to customer. After accepting customer requests, provider creates requested virtual machines with requested resources such as virtual processor type, speed, number of cores, memory size, storage capacity and network bandwidth. The provisioning can be done as advance provisioning, dynamic provisioning and user self-provisioning. In advance provisioning customer make contract with provider for specific services and provider does allocations with requested resources in advance to start the services, and for that customer will charge a flat fee or monthly bill in most of time. The best example for advance provisioning is requesting infrastructure for master database, which require higher CPU and memory. The dynamic provisioning is most common approach in industry as it is billed on pay per use basis. When customer needs, resources allocation is done, and when they don't require it de-allocates the resources. This feature support automatically in cloud providers like Amazon EC2 where it does the allocation and de-allocation based on demand of application load. The dynamic provisioning is also used as hybrid cloud private and public third-party cloud mix where customer get more flexibility over computing needs and cost. In self-provisioning, customer purchase resources through cloud provider's web interface by creating an account with payment details and requested resource will be provided to him within short time.

In this approach ansibile has been used for virtual system provisioning, and it may use vagrant for virtual machine load which is the simplest way of creating lightweight pre boot virtual servers for development environment. But when comes to industry, it may use modules like VMware vSphere, VirtualBox and AmazonEC2. Ansibile requires the information of network and IP ranges for inventory which is in YAML format and information of virtual server creation module, and then it creates custom cdrom image for ISO based provisioning or creates custom pre-boot execution environment (PXE) configuration for PXE based provisioning, then boot the system using ISO or PXE method to complete provisioning.

CHAPTER 4

DESIGN AND IMPLEMENTATION

4.1 Deployment Automation - CICD Pipeline

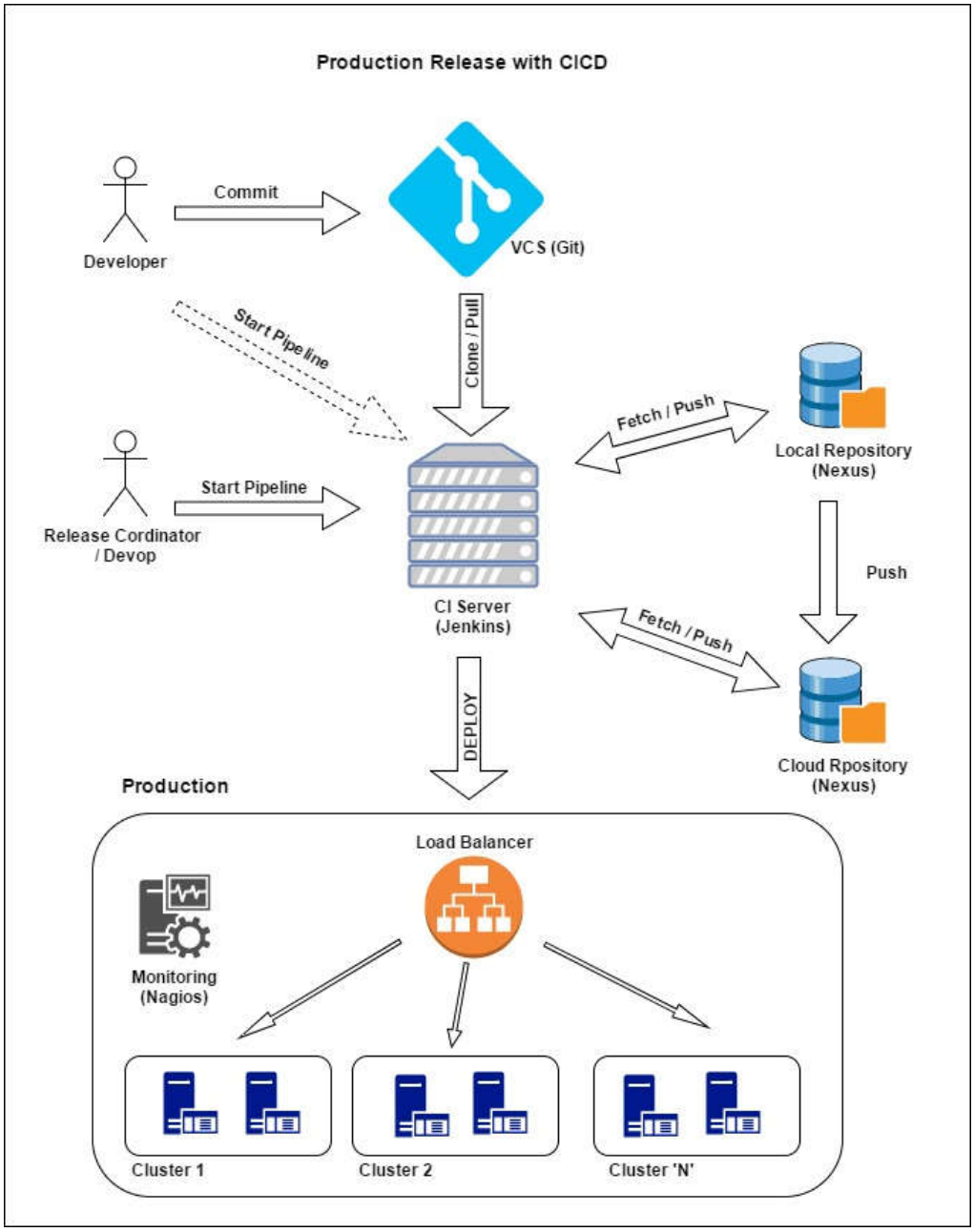


Figure 4. 1 The Production Release with CICD

The figure 4.1 shows the production release with CICD pipeline and it's a collaboration of various tools and services which perform as one unit in pipeline approach. In general agile approach, the pipeline is initiated by the release coordinator (RC), developer or Devop, but it's not restricted to them and it can be initiated by any member who has access to CI server and has rights to execute the relevant jobs. The developer commits all his changes into Git VCS using their feature branches. When one phase of development completed the all source codes are merged to release branch, and that release branch will be configured to build automatically by RC or senior developer. When there is sufficient local infrastructure, a team can have separate code branches which automatically build and deploy to separate local environment to improve the productivity and efficiency until it merge into the release branch. This build automation is done by CI server which is Jenkins in this approach. Jenkins can have separate jobs for each automation build process which can be configured to view as a group of branch wise like feature, develop, issue fix, or environment wise like Development, QA, UAT, Test Bench, Production or any other group of task wise like CICD deployment, test automation. When adding a new job to CI server, it clones the source code branch from Git by using one of Git user credential. The first build takes some time as it requires cloning all the source code from the given branch, but after that it only fetches changes from branch by using the pull request to Git branch. When production release the final build will be done by RC to have the latest changes in deployment artifact.

When completes the local automation build, the executable files and artifacts are pushed to local Nexus repository. There are several jobs which fetch those artifacts from Nexus when a team needs to deliver those artifacts to several environment through CI server instead of rebuild same source code multiple times. But when in production release having artifacts in two different network make deploying complex and it slows down as fetching data from local to production environments take time. To overcome that problem we can have another Nexus cloud repository which lies in same network. The relevant version artifacts will be pushed into this cloud repository through Jenkins by direct build or pushing local Nexus artifacts to Cloud. The figure 4.1 shows that the cloud CI server does push as well as fetch from the cloud Nexus,

but in production deployment Jenkins does not fetch or download artifacts to CI server, instead it directly fetch artifacts to application servers which are in the same network by automation Job scripts in Jenkins. The cloud Jenkins server executes the CICD pipeline automation Jobs to deploy the development into production servers, and when deployment happen, the nagios monitoring system is kept online to monitor any abnormal behavior throughout the process.

There is no specific rule or restriction on selecting tools, and as mentioned in chapter two and three the selection of these tools and deployment methods always depends on application. In here, there is a separate repository for keep third-party libraries and executable files, but another approach can be used Git for both purposes by creating another separate project for artifacts, which can be adopted for a small system. But when the system gets complex, the managing artifacts separately from source code is extended the features for automation process. In here, it has two Nexus repositories as local and cloud, but instead of having cloud Nexus another approach can have Artifact server which located in cloud same network, and it can use to do the same process in production release where all application servers get a copy from artifact server. When there is a separate Artifact server, instead of having one Jenkins server, having two Jenkins servers in local and cloud removes the network bandwidth issue. But having a cloud CI server is not necessary when there is cloud nexus, since application servers can directly download artifacts from cloud nexus. Therefore, this pipeline approach can scale based on requirement, where it can have separate VCSs, repositories, and CI servers to support the scale of the automation deployment process.

Automation Process

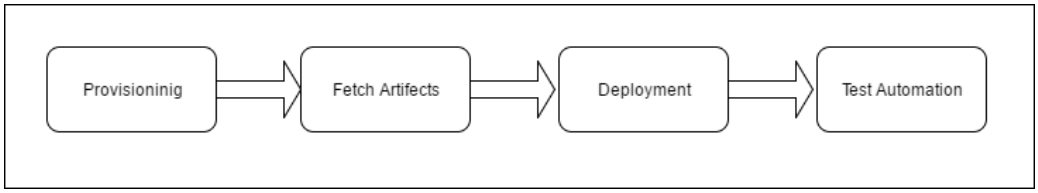


Figure 4. 2 The Automation Process Steps

As in figure 4.2, the main steps of deployment automation process are server provisioning, fetch Artifacts, Deployment and Test Automation. The provisioning means prepare the server available in a way that it is useful in production. When purchasing a cloud infrastructure, it has only the hardware specification that requested. Sometimes software installation part has to be done by the buyer or if required software available at a service provider, it can be purchased and can be installed on servers. Once the server provisioning completed, developed software which is called artifacts has to release into server. Artifacts can be an executable files/jar, database scripts, configuration files or any other form which completes the software development in production. There are several common practices of releasing executable files. One is, build online/real-time and do the deployment, this method require fast network connection and build should not take much time. Another way is, builds artifacts first and pushes them into predefined server (artifact server) for deployment. In this approach release team will be prepared the artifacts to be ready in previous day of deployment. Keeping all the artifacts in repository of same network is the best common practice in CICD approach. Deployable artifacts will automatically ready at repository when development push to UAT environment. Therefore, automation has to do fetch those artifacts into required server at deployment time. Deployment is the next phase of automation which has highest risk and gravity since all the changes are applied on live environment.

As in figure 4.3, the deployment starts with Artifact validation. The fetched artifact may have wrong software version, or pointed to a different location, also there can be permission issues in files which were fetched at early step. An artifact can be compressed files set or some configuration files or in any other format which is readable or measurable. Since most of deployment maintains the version, those artifacts can be validated by executing pre-prepared scripts which does tasks such as read the configuration files and validate configuration against server local parameters like internal IP address and ports, extract files and count number of files or validate file sizes in main folder, checking the directory structure of new deployment, and checking file permission. If any of the validation is failed, and then deployment has

to be stopped in that application server. Once the artifact verification completed, automation will initiate the draining of the load balancer traffic for the required server, as draining allows completing the existing sessions to finish transaction before completely stop the traffic. Once the drain message received by load balancer it stops the all the new incoming traffic to the server while it opens for responses to be completed. This makes the deployment smooth and unnoticed to users and they will never feel any disruption of service, and ultimately it gives zero downtime throughout the release.

Sometime this draining process takes more time than expected, as an example when a user has requested for long run transaction, it will hold the session without complete which makes automation to wait. As the automation can execute in parallel and its main goal is to complete the full server deployment faster, this waiting for server drain can pausing the automation. Because, though it capable of execute parallel, draining servers on same cluster can cause over load situation in remaining open servers of the cluster, as example if the cluster load balancer receives 20% of full load, and if there are five servers in a cluster to serve that load, that is 4% load for each, when drained one server then remain four will receive 5% load. When drained 2 servers then remain three opened servers will receive load of close to 7% on each, and if remain is two then each server receive 10% load. Sometimes holding a seven to ten percent load is possible with latest application server capabilities, but if that exceed the server limits, then it's become unavailable for the system, which introduce more load on remaining. Due to this risk most of deployments are happen at off-peak hours where it can cause minimum impact to system users. Therefore, automation has a configurable value of drain wait count, and once the count reach and it will re-enable the server by failing the new deployment on the server. This allows automation to proceed on next server and complete deployments faster as much as possible. When load balancer completed the server drain then automation starts to apply configurations such as soft link pointing on new software version, configuration file version replacement, property files replacement, and apply any connectivity changes like ha-proxy configuration. Then those configurations have to verify before restart nodes.

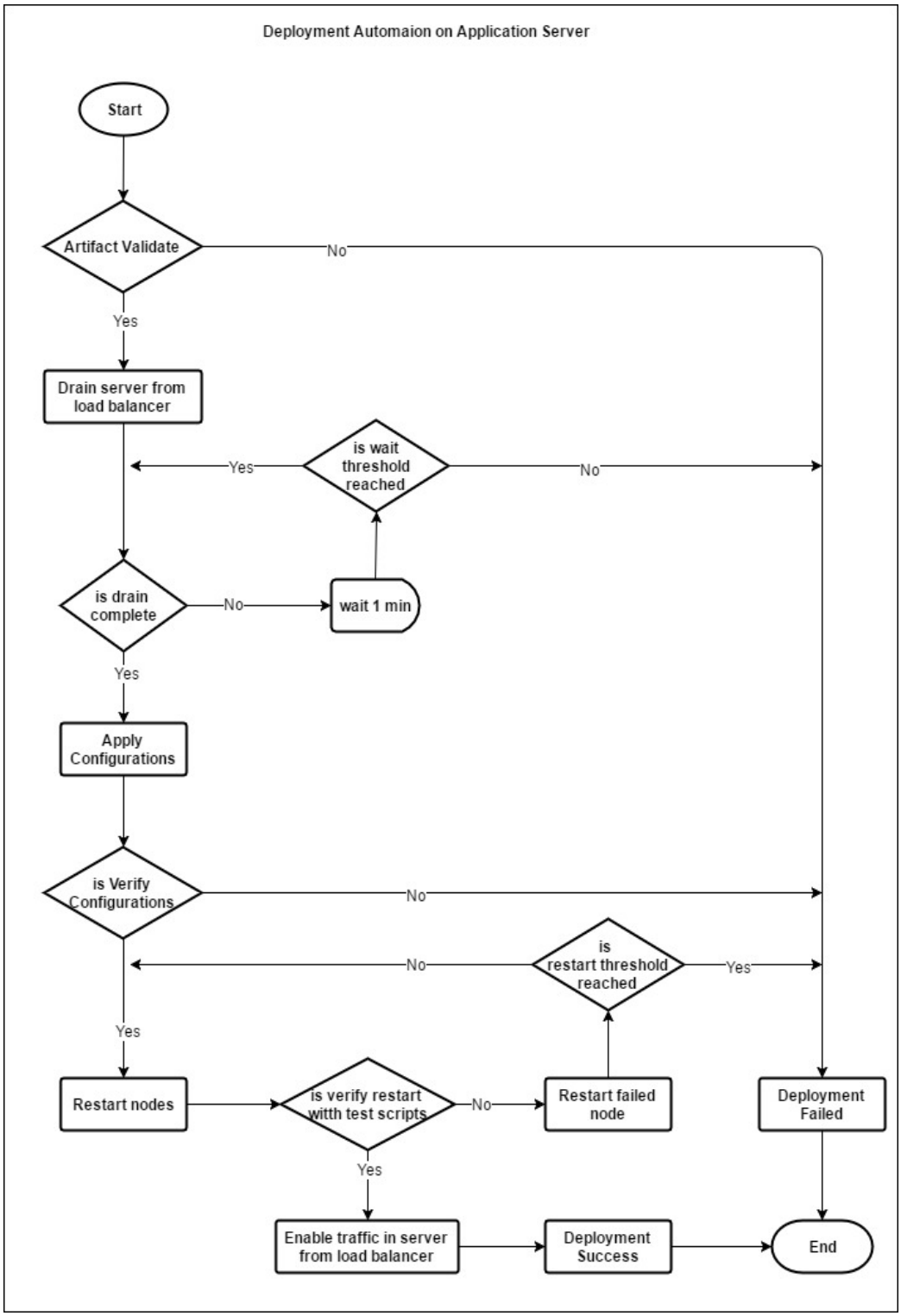


Figure 4. 3 The Deployment Automation on an Application Server

The automation can use simple scripts to validate soft links, software archive versions, and configuration files contents. When load balancer completed the server drain then automation starts to apply configurations. Then those configurations have to verify before restart nodes in same automation verification methods. When checking the new connectivity through proxies automation can restart server local proxies and verify the connectivity, and in case of common connectivity pools it can do graceful reload the pool or proxy and verify connectivity. When configuration verification fails it should not retry, as some cases it needs manual attention. In that case, automation should reconfigure to old version and deploy it with marked as failed deployment, and if that also fails, it should discard the server from automation process temporary until issue is fixed manually. The configuration verification has more gravity as it defines the consistency of system and it was the major drawback in manual deployment process. As example, after the deployment if one server node is still pointing to old version which is unnoticed as there are huge number of servers and nodes. If there are any node level configurations, it is important to verify each and every individual nodes as it hard to identify inconsistence results, due to it receives small fraction of system load.

To make configuration process fast and accurate the application must be automation deployment ready. It can be deployed executable files as one jar (fat jar), war or as ear archive instead of having several jar files in several locations, which make easy to upload and download and point soft links. One file for one version is more manageable and easy control with the automation. Next point is having centralized configuration or having common configuration file for connectivity, properties, and controllers for all nodes in a server. If there are four nodes in a server and if each node has separate configuration file, this makes more complex the automation process. In some legacy systems, it is hard to make the system as automation ready or CICD pipeline ready, since the effort it needs to remove those node level configuration introduce system level architecture changes. As example, some applications have hibernate database connectivity through pools which should be configured at node level to properly function, and removing that pool may introduce

software architecture change where it has to be used some other pooling mechanism or move to alternate database transaction method.

Once the configuration verification is completed then automation starts to restart nodes in server one by one and start the test automation in each node. This process can run in parallel to make it fast, but it can impact the system as all nodes tries to retrieve same resources at once. Most of the systems use cashed data or static data for speed up some processes, and those data completely load at node restart time from particular database or from another master cache server. As automation execute on parallel on server level, there is always set of servers getting apply the deployment. If automation deploys five servers at once and each server has four nodes, then those all twenty nodes starts to retrieve data from common resource is made overload situation and it can leads to service unavailability of that common resource and completely breaks the deployment. Therefore, unless if we aware that it does not impact to system, it should not execute parallel on node level. The test automation is to verify the basic functionality of system, and it can get failed at first few attempts due to reason of static data loading mentioned above. But this retry process can apply a threshold to avoid unnecessary restart attempt waiting, and if it reach that threshold then automation will stop the deployment on the server.

In practice, at the end of automations, there are more success deployments as well as few failed deployment servers due to reasons which explained above. Sometime those servers still in drain mode at load balancer level or some has enabled with old version. In this situation, the servers which are in drain mode has higher priority to fix since redeployment of old version servers can leads more servers unavailable to the system. In general rolling deployment method, team will complete deployment on one server and drain it and let it to run QA functional testing, and when the QA testing is completed full system deployment starts. When full system deployment complete, QA team can start their full system automation testing with basic functionalities, where test request are directed on system domain and request load are balanced. This test phase is optional, but it helps to identify any system misbehavior or inconsistency after the deployment.

4.2 Benchmark Automation

It is necessary to know current system benchmark level before release new version into production since it gives the comparison of performance between pre-state and post-state of the deployment. As in figure 4.4, the automation process is initiated by CI server and automation progress can be displayed through Jenkins job process output. It requires new benchmark server which is identical to production server which provisioned in same network environment, having equal hardware specifications and configuration, and using same type of connectivity for database and third-party services. As mentioned earlier, Go-replay has been used for traffic duplication purpose which is controlled by ansible automation and nagios is used for monitor behavior on benchmark server. Since this approach use live traffic for benchmark, it produce more accurate results than that simulated traffic and as go-replay can duplicate traffic without impact to host server, this approach is more safe. Also since go-replay can use to duplicate percentage level of original traffic we can get more clear idea of amount of load which breaks the system.

Benchmark Automation Steps:

1. Provision new server with homogeneous specs with same capacity as benchmark server.
2. Deploy current production version into benchmark server.
3. Initiate Go-replay from pre-selected app server list (Ex. App1)
4. Provide initial traffic from App1 starting with 5% load and gradually increase it with addition of 5% in each step with a gap of 5 minutes.
5. Check the nagios monitor status for latency of output result to see it reach defined expected maximum latency while service is available.
6. If benchmark server can perform when App1 traffic emission reach to 100%, then automation starts traffic duplication as in step 4 by selecting next App server to increase traffic load.

7. Continue steps 4, 5 and 6 until benchmark server breaks it limitation of latency or its service get unavailable.
8. Stop the traffic from all App servers when benchmark has reached its limit, and the load which breaks the server limit is the benchmark level of current production server.

In above the load which breaks the benchmark server limit is can be taken from multiple attempts and get average to have general production server benchmark level as there is a possibility of live load pattern can impact on this process. As example if we notice that most of the requests received at benchmark server during the benchmarking are high CPU consumed processes then that load can break the server limit easily and make service unavailable, and that service breaking load is not server average benchmark level. The load at breaking point can be noticed from nagios agent or from apache load balancer output traffic count. Instead of having traffic duplication from different servers, it is possible to have traffic duplication from one server while increase it more than 100% by copying the existing traffic, but that needs additional CPU cost for go-replay, which can cause impact on host application server, and also getting loads from different servers make traffic pattern receive at mean level which gives more accurate benchmark results.

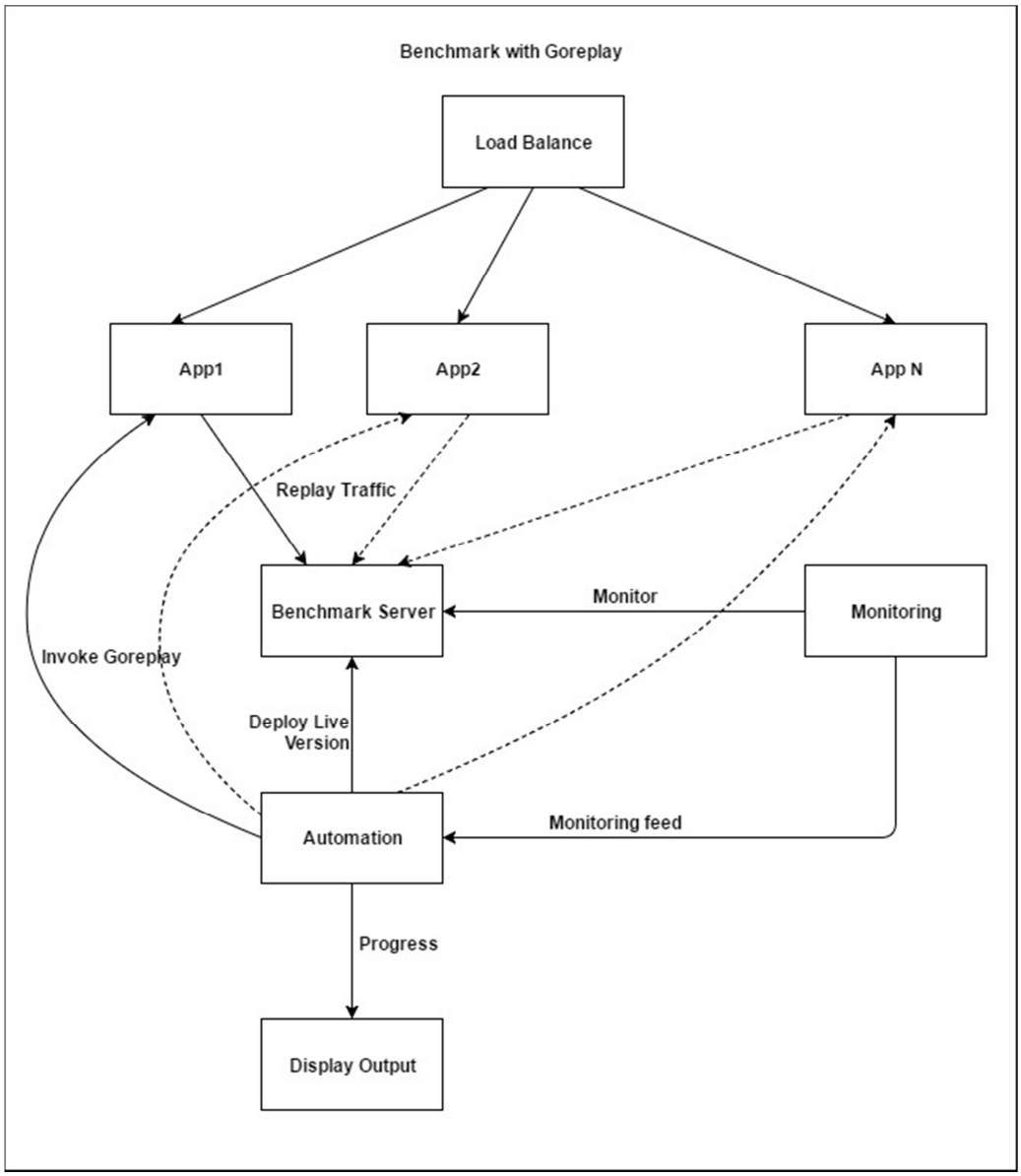


Figure 4. 4 The Benchmark with Go-replay

4.3 Load Test Automation

Once the benchmark automation is completed, the load test automation can be initiated by using same benchmark server or creating a new server by cloning a production as Load test server. Having separate load test server is better option than

using existing benchmark server, which requires to re-deploy and reconfigure the server in each time when it needs to switch between load test and benchmark, and also having separate load test server is reducing the complexity of deployment, and it can maintain clear separate automation scripts which allows to extend any feature as there is no dependency in environment. Once the load test server is available, the automation can be initiated using go-replay.

Load Test Automation Steps:

1. Provision new server with homogeneous specs with same capacity as load test server or use same benchmark server.
2. Deploy new development version into load test server.
3. Initiate Goreplay from pre-selected app server list (Ex. App1)
4. Provide initial traffic from App1 starting with 5% load and gradually increase it with addition of 5% in each step with a gap of 5 minutes.
5. Check the nagios monitor status for latency of output result to see it reach defined expected maximum latency while service is available.
6. If load test server can perform when App1 traffic emission reach to 100%, then automation starts traffic duplication as in step 4 by selecting next App server to increase the traffic load.
7. Continue steps 4, 5 and 6 until load test server breaks it limitation of latency.
8. Stop the traffic from all App servers when load test has reached its limit, and the load which breaks the server limit is the benchmark level of new development release.

In practice, since a new development phase adds more features into system it always tends to degrade the current performance of the system. As in same way of benchmark, in here also we can get limit breaking traffic load. But in here other than latency CPU, memory and thread pattern has to monitor more closely, because though it response with expected latency if the memory is at the edge level then there is a high chance it get service unavailable with production various traffic pattern. Therefore, having overall idea of all the load test monitoring parameters is vital before release the new software version into production.

4.4 Scaling Automation

The purpose of this automation is to identify new production server scale requirement and execute the necessary action of provision or de-provision of servers. Once we identified the production benchmark load and new development load, scaling requirement can be calculated as mentioned in section 3.4. But the process of de-provision at deployment time is not encouraging as new release need to be monitored for some time to see any misbehavior. Since load testing is performed for limited time frame for few hours, it may not reflect the actual peak load during the load test time, and de-provision is encouraging when system support automated demand scaling only, otherwise de-provision might introduce huge risk on production.

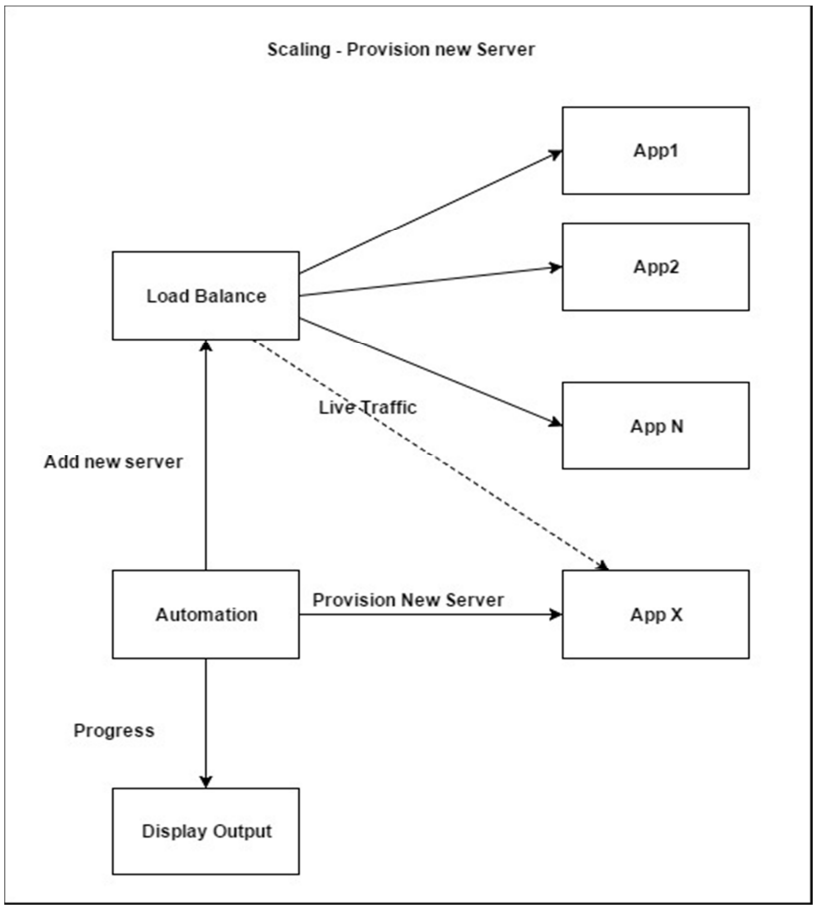


Figure 4. 5 The Scaling – New Server Provision

Scaling Automation Steps:

1. Calculate new server requirement as in section 3.4
2. Provision new servers with homogeneous specs with same capacity
3. Deploy current production version into new servers
4. Adding servers to load balancer with equal weight
5. Enable live traffic

In the step 3 we deployed the current production version to new servers since still new version has not been released into production and new servers should be enabled in production before the new version release, which shows in figure 4.5. Because, if new version release to live without having addition servers then there can be system performance issues until new server addition. The parallel provision of servers while deploying new version into production is an alternative option, but it also has risk if provision get delayed due some issues occurred at provision time such as provision verification failure, network issues and cloud providers' delay. In step 3, when adding new servers to load balancer it is assigned equal weight, but when consider a clustered environment, other than adding it to cluster load balancer with equal weight, it should alter the top level load balancer which control the load of the cluster and top level balancer has to be configured according to new proportionate of cluster server availability. Otherwise though it added to cluster load balancer, system is still suffering from under provision, since addition of new server has not applied on whole system and has not reduced the load overhead on other clusters.

CHAPTER 5

EVALUATION AND RESULTS

5.1 Test Bench Setup

Since this research work extends CICD pipeline approach, this section will focus on those extended features which evaluate through a test bench setup, and general CICD automation steps are omitted from this documentation. Once the CICD automation is done, this test bench setup has been initiated and automation modules for Test Bench are listed as in below table 5.1

Table 5. 1 The Test Bench Automation Modules

Module	Description
Server provisioning	Four application servers Nagios master server Load balancer server
Nagios setup	Configure nagios for all inventory/hosts in Master
Benchmarking	Initiate go-replay from app server to test server
Load testing	Deploy new application to test server and initiate go-replay from app server to test server
Scaling	provision new server and add it to load balancer

Though cloud virtualization is one of the evaluation environment option, for this approach the test bench is setup in local high capacity computer with local virtual instances. The details of all instances and installed software tools for this test bench are mentioned in table 5.2. The test bench setup is categorized into three phases as benchmark, load test and scaling. As in figure 5.1 four application servers are provisioned with tools mentioned in table 5.2 including nagios nrpe clients tools and provisioned separate nagios master server to monitor system, and APP4 is selected as

benchmark server. The HA proxy has been used as load balancer in testbench-lb server and Jmeter used to generate load directly on load balancer. Through automation, Go-replay initiated traffic replay from APP3 to APP4 then increase the load by starting replay from APP2 to APP4 while increasing the load up to 200%.

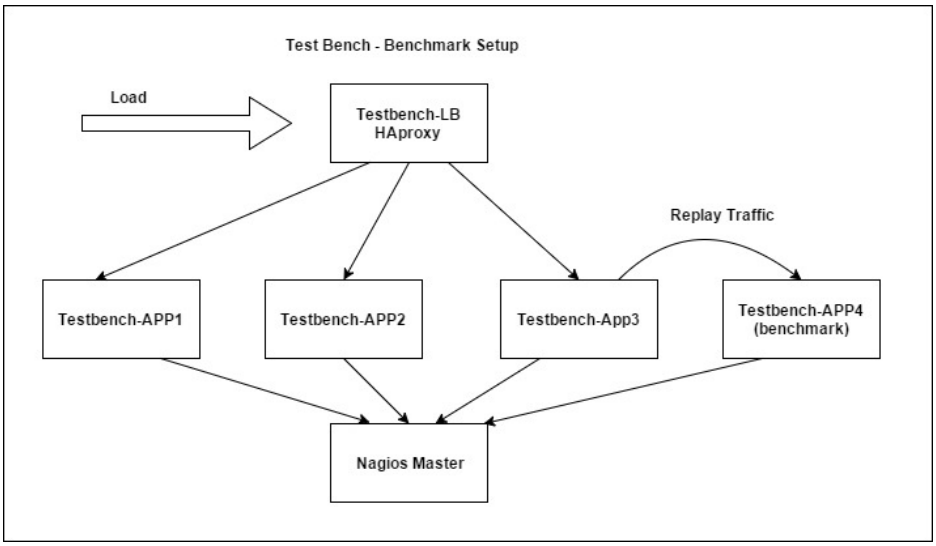


Figure 5. 1 Test Bench – Benchmark Setup

Once the benchmark phase completed, the same testbench-APP4 is used to perform load test phase. In here, first new version of application has to be deployed into APP4 through automation. Once deployment is completed, the load test process can be initiated which is equal to benchmark approach.

The application used in test bench approach is simple XML based application, which read XML requests, validate it and process requested dates rate details from cached objects and sent response in XML format. It does not use any database system, to reduce the complexity of this process and kept all rate details for next one year in CSV file. Application loads that data into cache in first attempt in row format then following next request will process those cache data to deliver response. It is introduced a discount application flow on rates as new version of the application by adding another CSV file which loads into cache in first attempt.

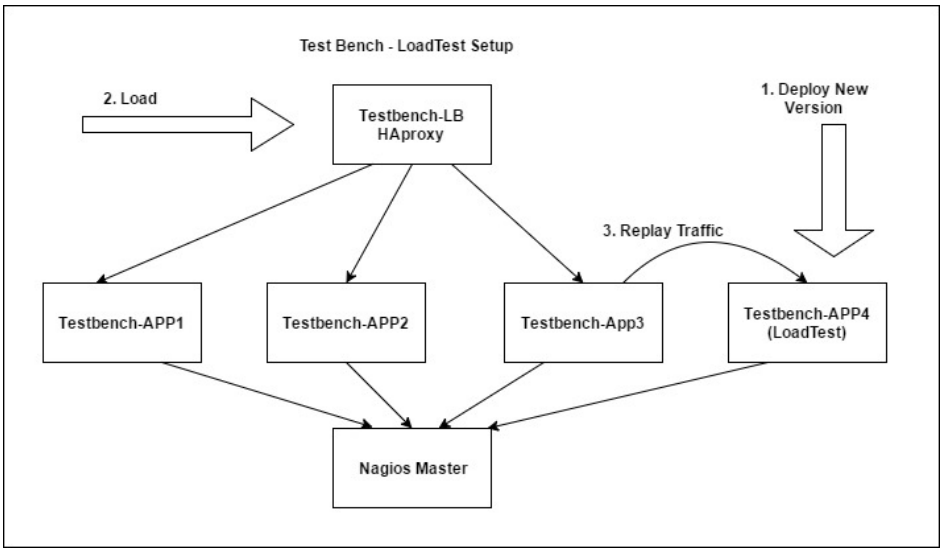


Figure 5. 2 Test Bench – Load Test setup

Table 5. 2 Test Bench Server Details

Server	**Details (Specification and Software tools)**
Nagios-master	• Single core 2GB server with Centos 7 – 64bit • Nagios core 4.1.1 and plugin 2.1.1
TestBench-APP1 to APP4	• Single core 2.5GB server with Centos 7 – 64bit • Java 8 • Tomcat 8.5 • Go-replay • Nagios nrpe -2.15
TestBench-LB	• Single core 1GB server with Centos 7 – 64bit • HA proxy 1.5.18 • Nagios nrpe -2.15

In scaling phase, the calculation of scale requirement needs amount of load at benchmark level and load test. Since APP4 getting requests from APP3, the proxy indicating traffic for APP3 incoming traffic is equivalent to APP4 load. In load

testing, the load amount is extracted from log file entry count. When scaling calculation is completed, if scaling requirement is identified then new server will be provisioned and included into load balancer for evaluate performance of system after scaled.

5.2 Test Bench Performance

As in test bench setup, the performance also can be described in benchmark, load test and scaling phases for clear understanding of the approach. In each phase how it should be performed, has been explained using nagios monitoring graphs with red color rectangle area that indicates specific cases.

5.2.1 Initial Bench mark Phase

In benchmark, APP4 which uses as benchmark server gets request from APP3 to have equal load, after some time it starts to get request from APP2 through go-replay to increase the load and find benchmark level. The figure 5.3 shows the APP2 CPU load and figure 5.4 shows the APP4 CPU load. The red color rectangular area shows the load increase state on benchmarking. When increase the traffic in APP4 its CPU load increased beyond critical level in figure 5.4, but in APP2 CPU has not showed that much of load and it is in below red line. This indicates when increase the load in application, the host server CPU also getting increase which can leads to service unavailability. In figure 5.5 and 5.6 shows free memory and memory usage behavior in APP2 and APP4 respectively. Unlike CPU load case, the memory has not shown any significant change in its behavior when increase the load as it utilize most of memory with proper application garbage collect processes, but having less memory also have risk of getting out of memory issues.

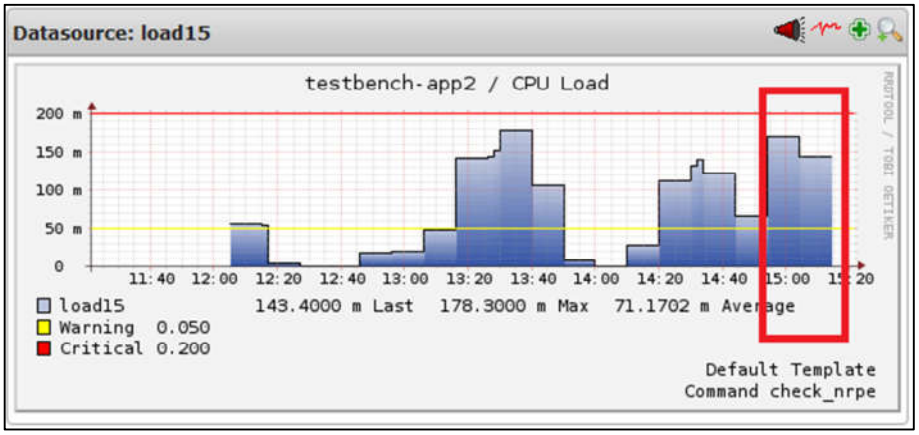


Figure 5. 3 Benchmark Phase – APP2 CPU load

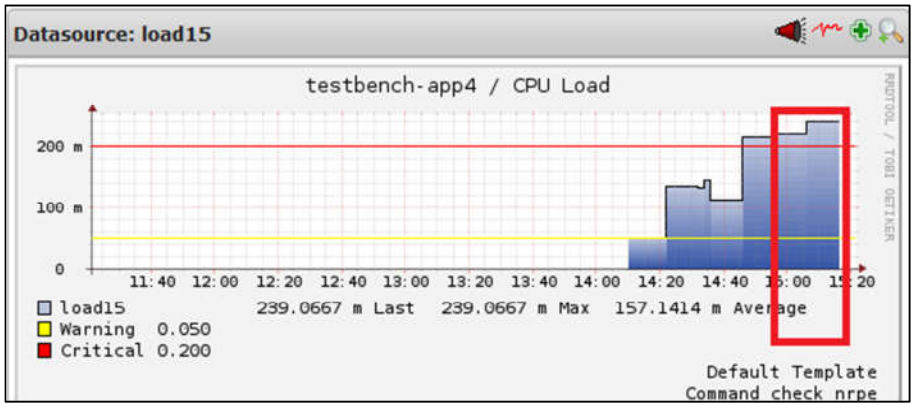


Figure 5. 4 Benchmark Phase – APP4 CPU load (Benchmark server)

To get the response time in the benchmark state, an additional Jmeter worker has been used which send requests on APP4 server in each 20th second and plotted the response time graphs. Since APP4 get one additional hit on every 20 second, it is negligible when compare to actual load on it, and since network is in the same local virtual environment, the network delay also negligible and as it is considered in all test bench phases its impact is equal in all phases. Figure 5.7 shows that response time graph of benchmarking at initial state. The graph shows high variation but it has maintained the response time around 55 to 72 milliseconds in most of time.

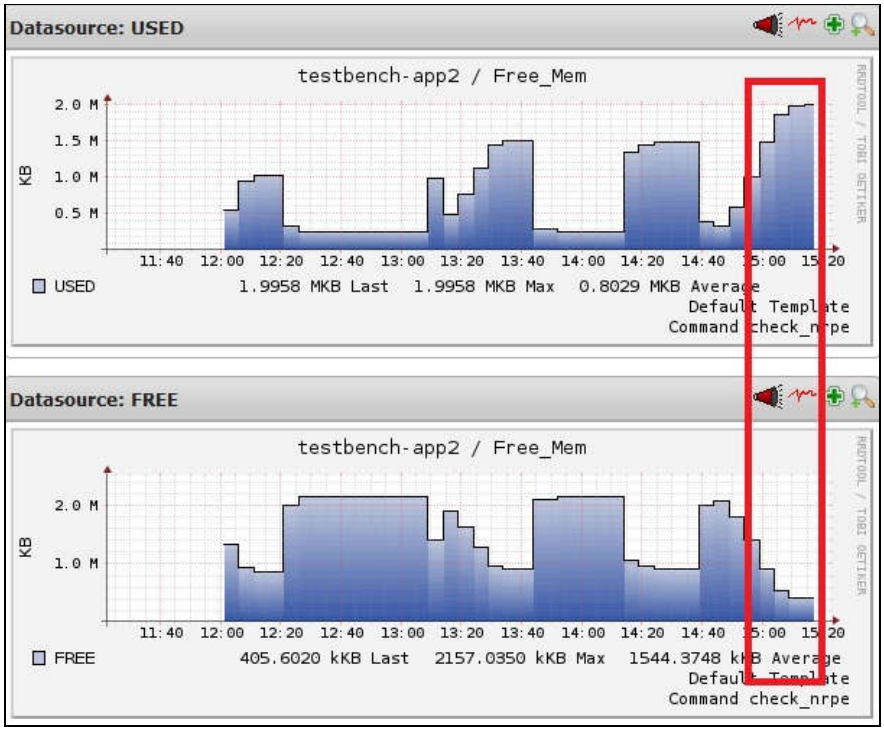


Figure 5. 5 Benchmark Phase – APP2 Free Memory

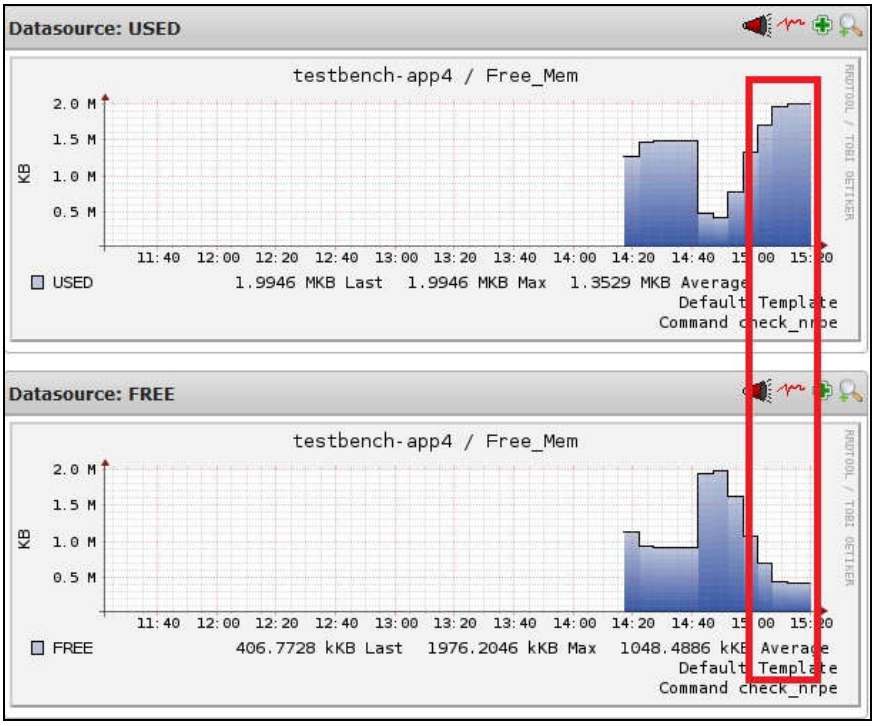


Figure 5. 6 Benchmark Phase – APP4 Free Memory (Benchmark server)

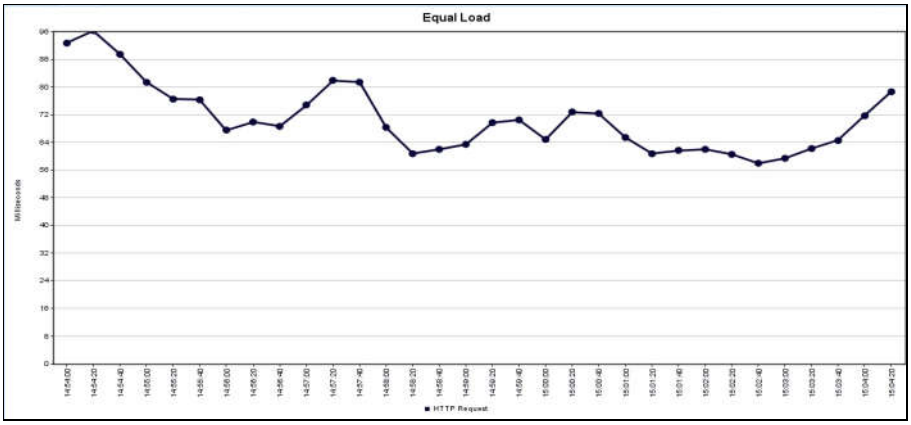


Figure 5. 7 Benchmark Phase – APP4 Initial Response Time Graph

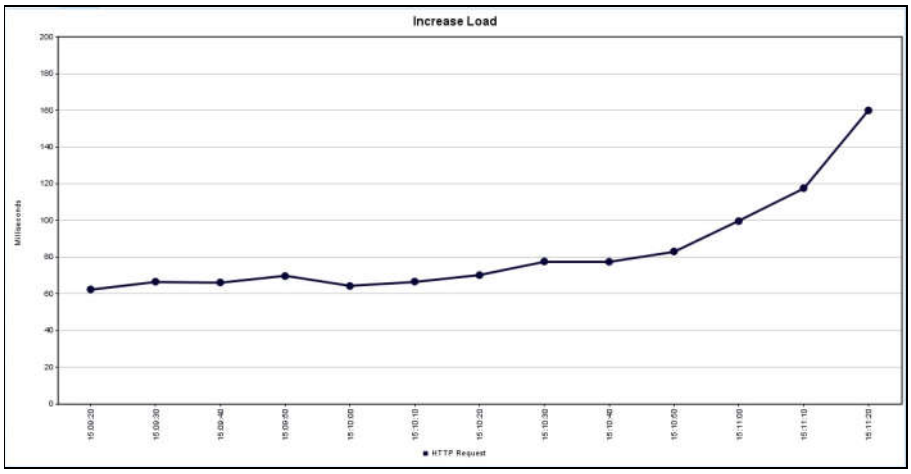
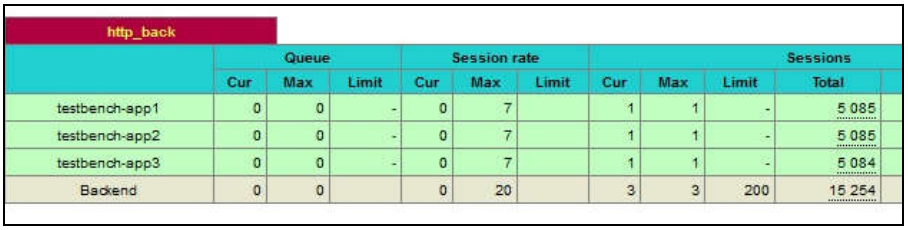


Figure 5. 8 Benchmark Phase – APP4 Increase Laod Response Time Graph

When APP4 load is increased, the response time has increased more than 100 milliseconds as in figure 5.8 which plotted using Jmeter worker 20 second ramp on APP4 server. Since application was getting out of memory issues more frequently while increasing load, it has to be plotted within short period to have clear variation. The benchmark load of the system is noted from load balancer HA proxy statistic report which shows each application server response count and total response. When APP4 response reached beyond 100 milliseconds, proxy statistic response count is taken as the bench mark load of the system, since in this test bench approach 100

milliseconds response time is considered as threshold response time of system which is application dependent value. As load balancer treat equally on all application servers, in this approach benchmark load is 5085 requests per APP and for duration of 20 minutes it totally responded 15254 requests as in figure 5.9.



http_back

	Queue			Session rate			Sessions			
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total
testbench-app1	0	0	-	0	7		1	1	-	5 085
testbench-app2	0	0	-	0	7		1	1	-	5 085
testbench-app3	0	0	-	0	7		1	1	-	5 084
Backend	0	0		0	20		3	3	200	15 254

Figure 5. 9 Benchmark Phase – APPs Total Response

5.2.2 Load Test Phase

When perform load test on the new version of software (in this case enabling new discount flow) as expected application response time has increased slightly, since addition of new discount flow was using CPU on processing rates. Though it had high variation in graph in normal flow in figure 5.7, it performed between 55 – 72 ms range. But in the load test, application performed in consistence level as in figure 5.10, at the response time between 70 – 80 ms which is slightly higher. The new version under performance was captured in nagios CPU load graph in figure 5.11. In that graph the time between 14.20 and 14.40 was initial benchmark level and then its load has increased when increasing benchmark load. The red rectangle is the phase of load test which is shown slight high CPU than that initial benchmark level, but it is less than that critical level. The impact of new version has also reflected in free memory graph in figure 5.12. It has taken more memory than initial benchmark phase as application loads discount data also into cache. Though memory usage increased, application has performed consistent since load was consistent, but working under low memory has high chances to give “out of memory” issues which were not experienced during load test phase. In this phase, it was counted 4117

requests were processed during 20 minutes which is less than that of benchmark level of 5085 requests processed. Therefore, it is necessary for check on scaling options in next phase before release new software version.

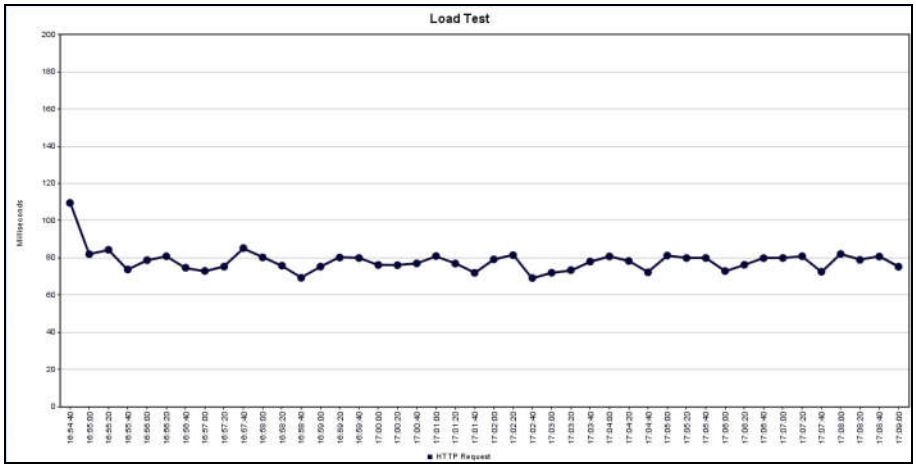


Figure 5. 10 – New Version Load Test Response Time Graph

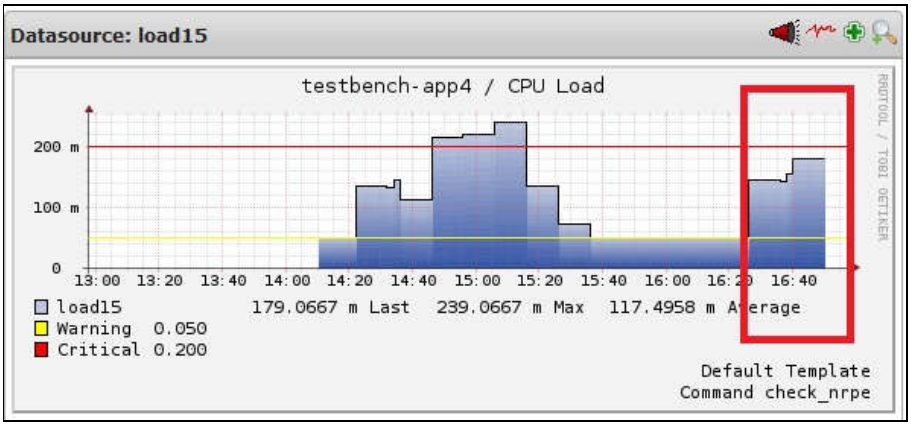


Figure 5. 11 Load Test Phase – CPU Load

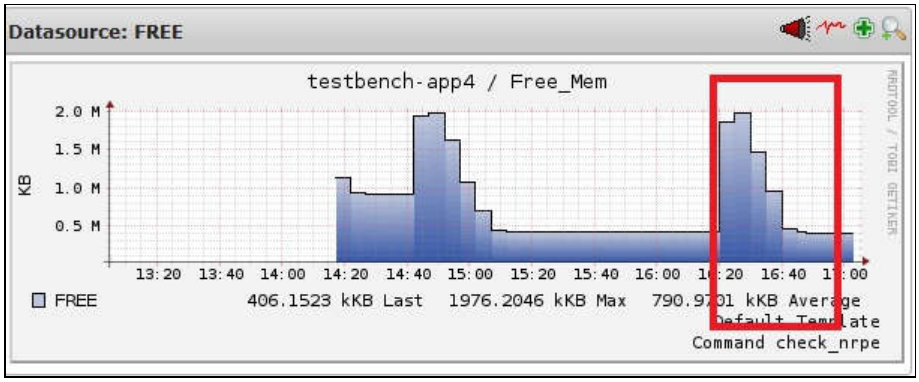


Figure 5. 12 Load Test Phase – Free Memory

5.2.3 Scaling Phase

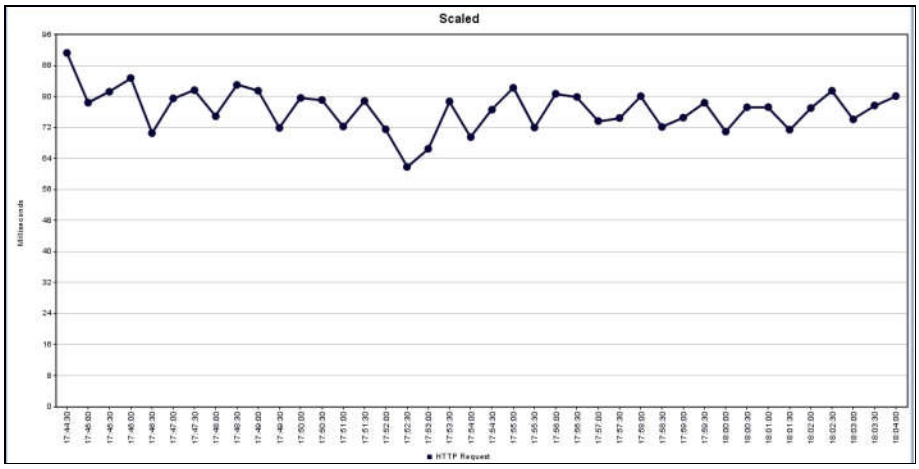


Figure 5. 13 After Scaled Response Time Graph

Scaling requirement calculation as in method explained in section 3.4:

Current benchmark load per server = Z

The maximum load could handle in load test = Z1

The total load of the system (number of nodes N) = Z x N

Load handling by new version in existing nodes = Z1 x N

Additional nodes requirement Zx = (N (Z – Z1))/ Z1

Since number of servers cannot be a decimal value, Zx should always return next (round to ceil) integer value from the result.

Z = 5085 Z1 = 4117

N = 3 (Three APP servers one node in each)

Zx = (N (Z – Z1))/ Z1

= (3 (5085 – 4117))/ 4117

= 0.71 => 1

Based on above calculation, the new server requirement is 1 additional server, to perform same as the expected benchmark level of system. Therefore, APP4 has been used as that additional server by including it on load balancer since both benchmark and load test phases are completed and APP4 is identical to other APP servers, as well as it is easy to reuse when need due to easy load balance configuration.

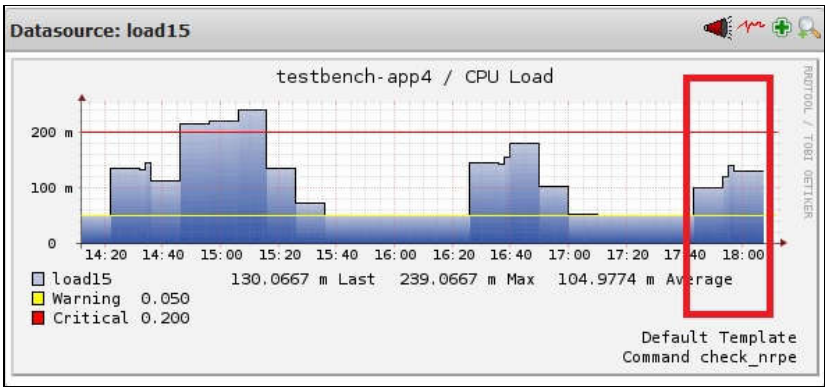


Figure 5. 14 Scaling Phase – APP4 CPU Load

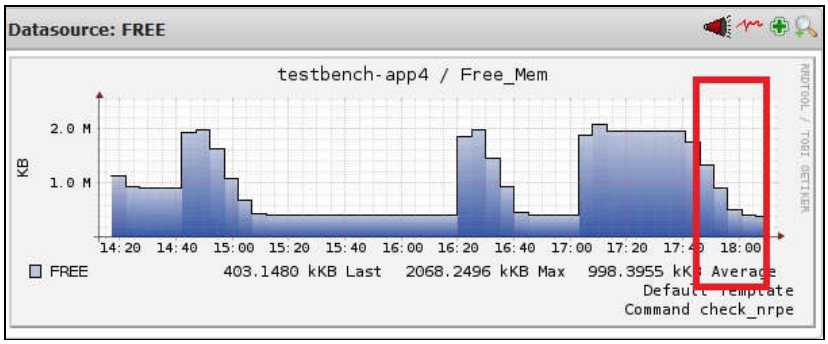


Figure 5. 15 Scaling Phase – APP4 Free Memory

http_back										
	Queue			Session rate			Sessions			
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total
testbench-app1	0	0	-	0	5		0	1	-	3 913
testbench-app2	0	0	-	0	6		0	1	-	3 912
testbench-app3	0	0	-	0	5		0	1	-	3 912
testbench-app4	0	0	-	0	5		0	1	-	3 912
Backend	0	0		0	21		0	1	200	15 649

Figure 5. 16 Scaling Phase – APPs Total Response

After adding APP4, system showed slight response time improvement as in figure 5.13 which shows better consistent pattern than initial benchmark. Though response time is higher than initial benchmark timing its most of responds were towards 70's ms range which is better than load test level. Since load is same and it is distributing among 4 servers, the CPU graph in figure 5.14 shows lower level than load test phase but memory graph in figure 5.15 has not changed significantly due to new version cache loading. If system has only three servers then it could have process only 12351 (load test 4117 x 3 severs) with new version, and initially it processed 15254 requests in benchmark phase. But after scaling as per figure 5.16, system has processed 15644 requests within same duration using four servers. The addition of new server is gained the benefit of consistent throughput, but it does not show much improvement in response time in system. Therefore, when the software is changing, the system is enabled to process same load through scaling automation, is the main fact which trying to prove through this test bench approach.

CHAPTER 6

CONCLUSION & FUTURE WORK

6.1 Conclusion

Due to agile time frame, releasing new software into production has more benefits than working on system optimizations. The scaling is one option which tries to evaluate through this process where software has tendency of making performance impact. There can be a situation where new software has to optimize than scale to have same or better performance, since scaling only allows processing more load but it does not fix the delays in response time.

CICD has made efficient agile delivery process as well as improved the productivity of the system. The proposed extended feature on CICD pipeline has achieved goals through test bench approach. It enables continuous benchmark of each software version and go-replay has given access to same live traffic on bench mark and load test phases. Ansible automation has enabled to perform test bench approach more efficient and addition of benchmark, load test and scaling phases on CICD has not introduced extra effort after initial setup. Test bench approach has proved that after scaling system could process same or better load with new development.

6.2 Study Limitations

- Simulated test data was not enough to cover all aspects.
- Application used cached data from local file instead of data from database system.
- Infrastructure was limited, as working on local virtual instance which bounded by computer capacity. Max memory used 2.5GB and only single core instance is

created. Due to that, maximum APP servers limited to 4, could not see any cluster behavior, and had to used simple rolling deployment strategy.

- Only used Performance parameters such as Response time, CPU load and Memory.
- The impact of caching on performance comparison is neglected due to assumption of equal hit ratio at both benchmark and load test phases.

6.2 Future Works

To avoid resource limitation test bench approach can be setup in cloud virtual servers and evaluate for better results. It considered response time on given load as main key performance parameter, but parameters such as CPU, memory, threads and IO can be used to calculation of scaling requirement which can predict more accurate scaling value. More CM tools like docker, puppet, chef and salt also can evaluate in CICD pipeline automation process to see which tool may simplify and faster in current approach.

Test bench application does not use any database system to reduce complexity in this approach, but most of systems have relational databases in their application, other than new static data structure. Therefore, evaluating the scaling option with database system is more realistic to current approach. The impact of application level cache on performance measurements can be evaluated by extending this approach for more accurate scaling. Also, this work could be extended to create self-healing elastic cloud approach.

REFERENCES

[1] N. Dzamashvili Fogelström, T. Gorschek, M. Svahnberg and P. Olsson, "The impact of agile principles on market-driven software product development", *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 22, no. 1, pp. 53-80, 2010.

[2] "Principles behind the Agile Manifesto", *Agilemanifesto.org*, 2017. [Online] Available: http://agilemanifesto.org/principles.html. [Accessed: Sep- 2017]

[3] H. Olsson and J. Bosch, "Climbing the "Stairway to Heaven" A multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software", in *38th Euromicro Conference on Software Engineering and Advanced Applications*, 2012

[4] M. Fowler, "Continuous Integration", *martinfowler.com*, 2017. [Online]. Available: http://martinfowler.com/articles/continuousIntegration.html. [Accessed: Sep- 2017].

[5] J. Humble and D. Farley, *"Continuous delivery : reliable software releases through build, test, and deployment automation"*, 1st ed. Addison-Wesley Professional, 2010.

[6] L. Chen, "Continuous Delivery: Huge Benefits, but Challenges Too", *IEEE Software*, vol. 32, no. 2, pp. 50-54, 2015.

[7] C. Dunlop and W. Ariola, "DevOps: Are You Pushing Bugs to Clients Faster?", Parasoft, 2015.

[8]S. Stolberg, "Enabling Agile Testing Through Continuous Integration", in *Agile Conference* 2009, pp 369-374.

[9] S. Kolli, "Automated Integration Testing & Continuous Integration for webMethods", CTO | CLOUDGEN, LLC, 2015

[10] M. Shahin, M. Ali Babar and L. Zhu, "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices", *IEEE Access*, vol. 5, pp. 3909-3943, 2017.

[11] A. Kumbhar, M. Shailaja and R. Anupindi, "*Getting started with Continuous Integration in Software Development*", Infosys Limited, 2015

[12] T. Lehtonen, S. Suonsyrjä, T. Kilamo, and T. Mikkonen, “Defining metrics for continuous delivery and deployment pipeline.” in *SPLST*, 2015, pp. 16–30

[13] *Practicing Continuous Integration and Continuous Delivery on AWS*. Amazon Web Services, Inc, 2017.

[14]T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm, “Continuous deployment at facebook and oanda,” in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE ’16. New York, NY, USA: ACM, 2016, pp. 21–30

[15] P. Suzie, "The Product Managers' Guide to Continuous Delivery and DevOps - Mind the Product", *Mind the Product*, 2016. [Online]. Available: https://www.mindtheproduct.com/2016/02/what-the-hell-are-ci-cd-and-devops-a-cheatsheet-for-the-rest-of-us/. [Accessed: Sep- 2017]

[16] A.Rahman, E. Helms, L. Williams, and C. Parnin. Synthesizing continuous deployment practices used in software development. In *Agile Conference (AGILE), 2015*, pages 1-10, Aug 2015.

[17] M. Hüttermann, *DevOps for developers*. [Berkeley, CA]: Apress, 2012.

[18] R. Seroter, "Exploring the ENTIRE DevOps Toolchain for (Cloud) Teams", *InfoQ*, 2017 [Online] Available: https://www.infoq.com/articles/devops-toolchain. [Accessed: Sep-2017]

[19] V. Pulkkinen, "Continuous Deployment of Software", *in Proc. Of the seminar no.58312107: Cloud-based Software Engineering*. University of Helsinki, 2013, pp 46-52

[20] S. Krusche and L. Alperowitz. Introduction of Continuous Delivery in Multi-Customer Project Courses, In *Proceedings of ICSE’14*. IEEE, 2014

[21] S. Krusche, L. Alperowitz, B. Bruegge, and M. Wagner. Rugby: An Agile Process Model Based on Continuous Delivery. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*. ACM 2014, 42–50.

[22] “Adopting Continuous Delivery” in *Continuous Delivery Whitepaper*. Levi9 IT Services, 2016.

[23] N. Dragoni, S. Dustdary, S. Larsenz and M. Mazzara, *Microservices: Migration of a Mission Critical System*. 2017.

[24] D. G. Feitelson, E. Frachtenburg, and K. L.Beck, “Development and Deployment at Facebook,” in *IEEE Internet Computing*, vol. 17, pp. 8- 17, July - August, 2013

[25] Mike Cohn., *Succeeding with Agile*. Pearson India, 2015, pp. 311-315.

[26] S.G.Gaikwad and M.A. Shah, "Pipeline Orchestration for Test Automation using Extended Buildbot Architecture", in *Confernce on Emerging Applications of Electronics System, Signal Processing and Computing Technologies (NCESC 2015)*, 2015.

[27] K. Alhamazani, R. Ranjan, K. Mitra, F. Rabhi, P. Jayaraman, S. Khan, A. Guabtni and V. Bhatnagar, "An overview of the commercial cloud monitoring tools: research dimensions, design issues, and state-of-the-art", *Computing*, vol. 97, no. 4, pp. 357-377, 2014.

[28] G. Aceto, A. Botta, W. de Donato and A. Pescapè, "Cloud monitoring: A survey", *Computer Networks*, vol. 57, no. 9, pp. 2093-2115, 2013.

[29] R. Khan, S. Ullah Khan, R. Zaheer and M. Inayatullah Babar, "An Efficient Network Monitoring and Management System", *International Journal of Information and Electronics Engineering*, 2013.

[30] M. A. Pervial “Using Nagios to monitor faults in a self-healing environment” *in seminar on Self-Healing Systems*. University of Helsinki, 2007

[31]M. C Montes, E. P Calle, F.J. R Calonge, “Using Nagios for intrusion detection”, *CHEP’04*, Interlaken, September 2004.

[32] J. Elmsheuser, A. Krasznahorkay, E. Obreshkov and A. Undrus, "A Roadmap to Continuous Integration for ATLAS Software Development", *Journal of Physics: Conference Series*, vol. 898, p. 072009, 2017.

[33] G. Fedoseev, A. Degtyarev, O. Iakushkina and V. Korkhov, "A continuous integration system for MPD Root: Deployment and setup in GitLab", SPbU, 2016.

[34] A. Balalaie, A. Heydarnoori and P. Jamshidi, "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture", *IEEE Software*, vol. 33, no. 3, pp. 42-52, 2016.

[35] D. Benavides, J. A. Galindo, Variability management in an unaware software product line company: an experience report, *in: The Eighth International Workshop on Variability Modelling of Softwareintensive Systems, VaMoS '14*, Sophia Antipolis, France, January 22-24, 2014.

[36] I. P. Barreiro, W. Booth, B. C. CERN Continuous Integration for Automated Code Generation Tools. *In: 14th International Conference on Accelerator & Large Experimental Physics Control Systems*, San Francisco, CA, USA, 6 - 11 Oct 2013

[37] C. MacNeill and S. Bodewig, "Apache Ant - Welcome", *Ant.apache.org*, 2017. [Online]. Available: http://ant.apache.org. [Accessed: Sep- 2017]

[38] B. Porter, J. Zyl and O. Lamy, "Maven – Welcome to Apache Maven", *Maven.apache.org*, 2017. [Online]. Available: https://maven.apache.org/. [Accessed: Sep- 2017].

[39] Rancher Labs, "Contnuous Integraton and Deployment with Docker and Rancher", Rancher Labs, 2016.

[40] "Gradle | Gradle vs Maven Comparison", *Gradle*, 2017. [Online]. Available: https://gradle.org/maven-vs-gradle. [Accessed: Sep- 2017].

[41] "Jenkins", *Jenkins*, 2017. [Online]. Available: https://jenkins.io. [Accessed: Sep- 2017].

[42] "7 Configuration Management (CM) Tools You Need to Know About", *Upguard.com*, 2017. [Online]. Available: https://www.upguard.com/articles/the-7-configuration-management-tools-you-need-to-know. [Accessed: Sep- 2017].

[43] "Compute Engine Management with Puppet, Chef, Salt, and Ansible | Solutions | Google Cloud Platform", *Google Cloud Platform*, 2017. [Online]. Available: https://cloud.google.com/solutions/google-compute-engine-management-puppet-chef-salt-ansible. [Accessed: Sep- 2017].

[44] V. Hardion, D. Spruce, M. Lindberg, A.M. Otero, J.Simon, J. Jamroz, A. Persson, “Configuration Management of the control system”. *in ICALEPCS2013,* San Francisco, 2013

[45] E. Afgan, K. Krampis, N. Goonasekera, K. Skala, and J. Taylor, “Building and provisioning bioinformatics environments on public and private clouds”, *in MIPRO 15: 38th International Convention on Information and Communication Technology, Electronics and Microelectronics*, May 2015, pp. 223–228.

[46] F. C. Liu, F. Shen, D. H. Chau, N. Bright, and M. Belgin, “Building a research data science platform from industrial machines,” *3rd Workshop on Advances in Software and Hardware for Big Data to Knowledge Discovery (ASH) co-located with IEEE Big Data Conference*, 2016.

[47] "GoReplay - test your system with real data", *Goreplay.org*, 2017. [Online]. Available: https://goreplay.org. [Accessed: Sep- 2017].

[48] Z. Li, L. O’Brien, H. Zhang, R. Cai, “On a catalogue of metrics for evaluating commercial cloud services”, *in: Proceedings of the ACM/IEEE 13th International Conference on Grid Computing (GRID)*, 2012, pp. 164 –173

[49] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears, “Benchmarking Cloud Serving Systems with YCSB”, *ACM Symposium on Cloud Computing* (SoCC), Indianapolis, Indiana, June 2010.

[50] K. Hwang, X. Bai, Y. Shi, M. Li, W. Chen and Y. Wu, "Cloud Performance Modeling with Benchmark Evaluation of Elastic Scaling Strategies", *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 1, pp. 130-143, 2016.

[51] N. R. Herbst, S. Kounev and R. Reussner, "Elasticity in Cloud Computing: What It Is, and What It Is Not," in *ICAC,* 2013. pp. 23–27

[52] E. Barrett, E. Howley and J. Duggan, "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud", *Concurrency and Computation: Practice and Experience*, vol. 25, no. 12, pp. 1656-1674, 2012.