Automating Infrastructure as a Code using Continuous Integration and Continuous Delivery

Bhanupriya H, Dr. Krishna A N, Ravi Shekhar Jha
PG student[1], Professor [2], DevOps Specialist [3],
Department of Computer Science, SJBIT, under VTU, Bangalore-60
bhanupriya15393@gmail.com, csehod@sjbit.edu.in, ravisjha@hpe.com

ABSTRACT-Every project team expects their project management to be automated and robust. With advancements in automation technologies, one can think of automating the manual built infrastructures like storage servers, project servers and deployments on those servers. We can achieve this automation of Infrastructure and build a continuous delivery pipeline which accelerates the speed of project development, testing and deployment using Emerging technologies. This paper depicts how team can leverage the Continuous Integration and Continuous Delivery concepts to develop their projects to increase the efficiency and speed up their releases.

Keywords - Configuration management, Continuous Delivery, Continuous Integration, GitHub, Jenkins

1.Introduction

Software Release In Organizations To Users Is Often Painful, Risky And Time Consuming. Continuous Integration And Continuous Delivery (Ci/Cd) Can Help Big Organizations Become As Slender, Agile And Innovative As Startups. Through Trustworthy, Low Risk Releases Ci/Cd Makes It Possible To Continuously Adapt Software Aligned With User Feedback, Market Shifts And Changes In Business Strategy [1].Test, Support, Development And Operations Work Together As One Delivery Team To Automate And Simplify The Build, Test And Release Process. Continuous Integration Is Outlined As Fundamental Practice In Extreme Programming Methodology, It Has Emerged As An Essential Element For Teams Doing Iterative And Incremental Software Delivery.Continuous Delivery Is Extension Of Continuous Integration, Which Ensures The Team That Every Change They Make To The System Will Be Releasable, And That We Can Release Any Version At Just Push Of A Button.

2. Problem Statement

Having a bad development workflow will be costly; it degrades the productivity of engineers in development to deployment cycles. A great workflow can make any good developer to be great and best ones to be exceptional. The most important problem that we face as software professionals is: If somebody thinks of a best idea, how do we deliver it to users as quickly as possible.

There are many software development methodologies which primarily focus on requirement management and its effect on the development effort. It is challenging to find what happens once requirements are identified, solutions formulated, developed and tested, how these activities joined together and synchronized to make process as efficient and reliable as team can make it? How do we

able developers, testers, and build and operation engineers to work together effectively?

The day of a software release tends to be a tense one. The process used to make the release of projects increases the degree of risk and terrifying sometimes. In many software projects, release is a manually intensive process; finally the application is started, piece by piece if it's a distributed or service oriented application. If any step is not perfectly executed, the application won't run properly. It is very difficult to identify what went wrong and where the error is.

Disadvantages of manual deployment:

The creation of extensive, detailed documentation that describes the steps to be taken and the ways in which the steps may go wrong during deployments. Confidence on manual testing to confirm that the application is running correctly. Repeated calls to the development team to explain why a deployment is going wrong on a release day. Frequent rectifications to the release process during the course of a release. Releases that take more than a few minutes to achieve. Releases that are unpredictable in their result to be rolled back.

Need for Automated testing & deployment

When deployments aren't fully automated, errors will occur every time they are executed. The question of interest is whether or not the errors are noteworthy. Even with excellent deployment tests, bugs can be hard to track down.

Automated deployments encourage cooperation, because everything is explicit in a script. Documentation has to make assumptions about the level of knowledge of the reader and in reality is usually written as a reference for the person performing the deployment, making it solid to others.

Risk in deploying to production like environment

Releasing into staging is the first time that operations team interact with the new release. In some organizations,

separate operations teams are used to deploy the software into staging and production. In this case, the first time an operations person sees the software is the day it is released into production. Either a production-like environment is costly enough that access to it is strictly controlled.

The development team assembles the correct installers, configuration files, database migrations, and deployment documentation to pass to the people who perform the actual deployment—all of it untested in an environment that looks like production or staging.

When the deployment to staging occurs, a team is assembled to accomplish it. Sometimes this team has all the necessary skills, but often in very large organizations the responsibilities for deployment are divided between several groups and it results in poor collaboration. It should also be possible to use the same automated process to roll back to a previous version of production if the deployment goes wrong.

3. PROPOSED METHOD

The Software release should be a low-risk, frequent, cheap, rapid, and predictable process. Our goal is to describe the use of deployment pipelines, combined with high levels of automation of both testing and deployment and comprehensive configuration management to deliver push-button software releases.

Every Change Should Trigger the Feedback Process

An operational software application can be usefully decomposed into four components: executable code, configuration, host environment, and data [2]. If any of them changes, it can lead to a change in the behavior of the application. Therefore we need to keep all four of these components under control and ensure that a change in any one of them is tested.

Executable code changes when a change is made to the source code. Every time a change is made to the source code, the resulting binary must be built and tested. In order to gain control over this process, building and testing the binary should be automated. Continuous integration is the practice of building and testing your application on every check-in.

This executable code should be the same operational code that is deployed into every environment, whether it is a testing environment or a production environment. Anything that changes between environments should be noted as configuration information. Any change to an application‘s configuration, in whichever environment, should be tested.

The tests will vary depending on the system, but they will usually include at least the following checks. The process of creating the executable code must work. This verifies that the syntax of your source code is valid. The software‘s unit tests must pass. This checks that your application‘s code behaves as expected. The software should fulfill certain quality criteria such as test coverage and other technology-specific metrics.

The software‘s functional acceptance tests must pass. This checks that your application conforms to its business acceptance criteria—that it delivers the business value that was intended. The software‘s nonfunctional tests must pass. This checks that the application performs sufficiently well in terms of capacity, availability, and security, and so on to meet its users‘ needs.

The software must go through empirical testing and a demonstration to the customer and a selection of users. This is typically done from a manual testing environment. In this part of the process, the product owner might decide that there are missing features, or we might find bugs that require fixing and automated tests that need creating to prevent regressions.

Open to feedbacks in early stages

The key to fast feedback is automation. With fully automated processes, your only limitation is the amount of hardware that you are able to throw at the problem. If you have manual processes, you are dependent on folks to get the job done. People take longer, they introduce errors, and they are not auditable. Moreover, performing manual build, test, and deployment processes is boring and repetitive—far from the best use of people. Developers should commit changes to their version control system frequently, and fragmented code into separate components as a way of managing large or distributed teams.

The Delivery Team must be reactive to Feedback

It is essential that everybody involved in the process of delivering software is involved in the feedback process. That includes developers, testers, operations staff, database administrators, infrastructure specialists, and managers.

A process based on continuous improvement is essential to the rapid delivery of eminent software. Iterative processes help establish a regular heartbeat for this kind of activity—at least once per iteration a retrospective meeting is held where everybody discusses how to improve the delivery process for the next iteration.

Finally, feedback is not noble unless it is acted upon. This requires discipline and planning. When something needs doing, it is the responsibility of the whole team to stop what they are doing and decide on a course of action. Only once this is done should the team carry on with their work.

Scaling of the process

One common complaint we hear is that the process we describe is idealistic. It may work in small teams, these critics say, but it can‘t possibly work in any huge, distributed project. Both the theory and the practice are as relevant to large teams as they are too small, and our experience has been that they work.

4. METHODOLOGY

Create a Repeatable, Reliable Process for Releasing Software

Releasing software should be easy. It should be easy because you have tested every single part of the release process hundreds of times already. It should be as modest as pressing a button. The repeatability and reliability derive from two principles: automate almost everything, and keep everything you need to build, deploy, test, and release your application in version control.

Deploying software eventually involves three things: 1) Provisioning and managing the environment in which your application will run (hardware configuration, software, infrastructure, and external services). 2) Installing the correct version of your application into it. 3) Configuring your application, including any data or state it requires.

The deployment of your application can be realized using a fully automated process from version control. Application configuration can also be a fully automated process, with the necessary scripts and state kept in version control or databases. Clearly, hardware cannot be kept in version control; but, particularly with the advent of cheap virtualization technology and tools like chef, ansible, the provisioning process can also be fully automated [3].

Automate Almost Everything

There are some things it is impossible to automate. Exploratory testing depend on experienced testers. In general, your build process should be automated up to the point where it needs specific human direction or decision making. This is also true of your deployment process and, in fact, our entire software release procedure.

Automation is a prerequisite for the deployment pipeline, because it is only through automation that we can guarantee that people will get what they need at the push of a button. However, you don‘t need to automate everything at once. You should start by looking at that part of your build, deploy, test, and release process that is currently the tailback. You can, and should, automate gradually over time.

Keep Everything in Version Control

Everything you need to build, deploy, test, and release your application should be kept in some form of versioned storage. All of the necessary stuff should be version-controlled, and the relevant version should be identifiable for any given build. That is, these change sets should have a single identifier, such as a build number or a version control change set number that references every piece.

It should be possible for a new team member to sit down at a new workstation, check out the project‘s revision control repository, and run a single command to build and deploy the application to any accessible environment, including the local development workstation. It should also be possible to see which build of your various applications is deployed into each of your environments, and which versions in version control these builds came from.

If it’s risky test it more rather than at later stage

This is the most general principle on our list, and could perhaps best be described as a heuristic. Integration is often a very painful process. If this is true on your project, integrate every time somebody checks in, and do it from the start of the project.

If releasing software is painful, aim to release it every time somebody checks in a change that passes all the automated tests. If you can‘t release it to real users upon every change, release it to a production-like environment upon every check-in. If creating application documentation is painful, do it as you develop new features instead of leaving it to the end. Extreme programming is essentially the result of applying this heuristic to the software development process. Much of the advice in this book comes from our experience of applying the same principle to the process of releasing software [4].

Build Quality In

The earlier you catch defects, the cheaper they are to fix. Defects are fixed most cheaply if they are never checked in to version control in the first place. Firstly, testing is not a phase, and certainly not one to begin after the development phase. If testing is left to the end, it will be too late. There will be no time to fix the defects. Secondly, testing is also not the domain, purely or even principally, of testers. Everybody on the delivery team is responsible for the quality of the application all the time.

Done Means Released

For some agile delivery teams, ―done‖ means released into production. This is the ideal situation for a software development project. However, it is not always practical to use this as a measure of done. The initial release of a software system can take a while before it is in a state where real external users are getting benefit from it. So we will choose the next best option and say that a functionality is ―done‖ once it has been successfully showcased, that is, demonstrated to, and tried by,

representatives of the user community, from a production-like environment.

Start by getting everybody involved in the delivery process together from the start of a new project, and ensure that they have an opportunity to communicate on a recurrent regular basis. Once the barriers are down, this communication should occur continuously, but you may need to move towards that goal incrementally. This is one of the central principles of the DevOps movement [7].

Continuous Improvement

It is worth highlighting that the first release of an application is just the first stage in its life. All applications evolve, and more releases will follow. It is important that your delivery process also evolves with it.

The whole team should regularly gather together and hold a retrospective on the delivery process. Somebody should be nominated to own each idea and ensure that it is acted upon. Then, the next time that the team gathers, they should report back on what happened. This is known as the Deming cycle: plan, do, study, and act [8].

5. TOOLS USED

This section demonstrates how we can build Continuous Integration and Continuous Delivery pipeline and thus find the solution to the problem explained in 2nd section above. We need Version control systems (Source code Management), Build automation server (Jenkins), Configuration management tool (chef), Selenium (testing scripts), Servers for deployment at different environments (Development, Staging, Production) and any project code that has to be deployed residing in SCM (Source Code Management). We will design CI/CD (Continuous Integration and Continuous Delivery) pipeline after going through each of these selected tools.

Distributed Version control systems

The rise of distributed version control systems (DVCSs) is transforming the way teams cooperate. Where open source projects once emailed patches or posted them on forums, tools like Git make it extremely easy to pull patches back and forth between developers and teams and to branch and merge work streams. DVCSs allow you to work easily offline, commit changes locally, and rebase or defer them before pushing them to other users.

The core characteristics of a DVCS is that every repository contains the entire history of the project. GitHub has an additional layer of indirection: Changes to your local working copy must be checked in to your local repository before they can be pushed to other repositories, and updates from other repositories must be resolved with your local repository before you can update your working copy.

In the traditional model, committers acted as gatekeepers to the definitive repository for a project, accepting or rejecting patches from contributors. Due to the various above advantages we choose GitHub as version control system for building CI/CD pipeline.

Build automation server- Jenkins

Jenkins has a large pool of plugins allowing it to integrate with pretty much every tool in the build and deployment ecosystem. Once CI (Continuous Integration) tool jenkins is installed, it should be possible to get started in just a few minutes by telling your tool where to find your source control repository, what script to run in order to compile, if necessary, and run the automated commit tests for your application, and how to tell you if the last set of changes broke the software. It is general purpose job executer.

Configuration management tool-Chef

There are a number of solutions for managing operating system configuration, including which software and updates are installed, on an ongoing basis. Perhaps the most popular are ansible, Puppet, and Chef. We have used Chef as the tool for building CI/CD pipeline in this paper, it is a powerful automation platform that transforms complex infrastructure into code, bringing your servers and services to life[3].

Chef is built around simple concepts: achieving desired state, centralized modeling of IT infrastructure, and resource primitives that serve as building blocks. These very same concepts allow Chef to handle the most difficult infrastructure challenges on the planet. Anything that can run the chef-client can be managed by Chef. Chef recipes are the programs written in ruby used to achieve the desired configuration. Need to bootstrap the nodes/servers which has to be configured only the first time to establish connection between client and chef server. Figure1 shows the client server architecture of chef topology.

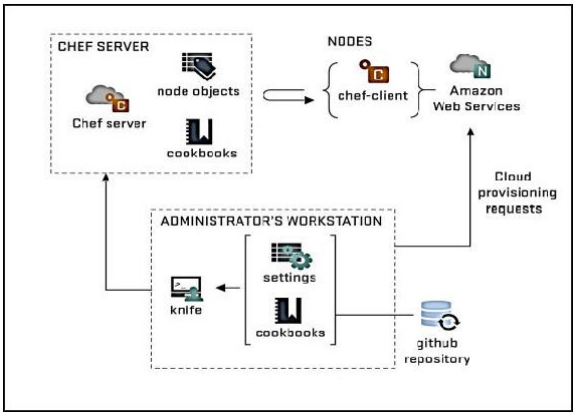

Figure 1: Chef Topology

Selenium scripts for testing

Selenium is a set of different software tools each with a different approach for supporting test automation. Most Selenium QA Engineers focus on the one or two tools that most meet the needs of their project, however learning all the tools will give you many different options for approaching different test automation problems.

Deployment servers

These are the servers used to deploy the code in different environments like Development where developers see how changes look like, Staging where testing can be done and Production where users are allowed to access once the application is released [12].

6. PROCESS

We can setup CI/CD pipeline using the above tools. For using Chef we need chef Development kit to program the necessary configurations. Chef Server has to be setup on Red Hat or any OS. It requires bootstrapping of nodes/servers (here its deployment servers) for the first time to establish connection between the server and those deployment servers which acts as client to the chef server. Configuration includes installation of necessary packages, setting environment variables to facilitate running of the application. Recipes are the program consisting of configurations necessary for deploying. It is uploaded as cookbooks, data bags, roles etc. on Chef Server which is later made to execute on Clients bootstrapped. These are applied to clients (servers) when we run the command –chef-client‖ on those servers. Thus Chef used for deployment on servers.

In Jenkins, jobs can be configured such a way that it pulls the application code that has to be deployed from the GitHub when a change is made to it and run the chef command –chef-client‖ on the server on which it has to be deployed i.e. development server. Job can also be configured/created such a way that deployment to staging can happen every night and tested every night each changes made and results are sent to the developers as a feedback in the form of an email. By this, we can configure multiple jobs to perform deployments on servers on different environment, testing of those applications and set to run on conditional basis.

Whenever the developer checks in the code to GitHub it creates a pull request and send a message to Jenkins to deploy code on development server so that developer can see the effect of changes he made to the application codebase. Build happens on development and notifies developer if deployment was successful on completion of executing –chef-client‖ on that server.

Every night the changes made to the application to be deployed in GitHub is deployed automatically to the staging server and selenium scripts are made to run on that staging environment to perform testing of the application. Results are sent via email to the developers as a feedback to rectify mistakes/errors if any. This is called Nightly build.

Once the development team feels the application is ready to release without any errors, code from GitHub can be deployed to production server where end users can access it. By this way, releases are made easy, less time consuming and robust. This can boost efficiency of development team and the business strategy.

Figure 2 shows the Deployment workflow cycle, Figure 3 shows the overview of how the pipeline can be setup. Figure 4 shows the feedback mechanism in CI/CD. Figure 4 shows the GitHub overview of how application code that has to be deployed looks like. Figure 6 is snapshot of chef server where we can manage cookbooks that is run on servers. Figure 7 shows the Jenkins setup that is necessary to setup automatic jobs to deploy automatically whenever necessary.

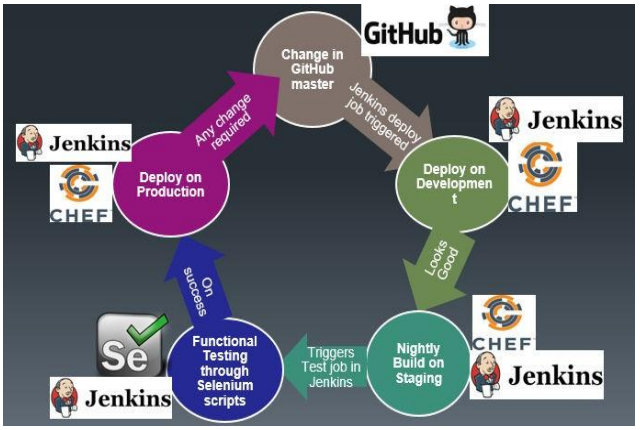

Figure 2: Workflow of Deployment Cycle

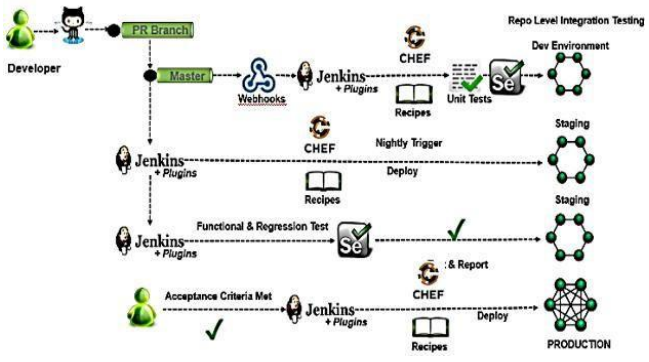

Figure 3: Continuous Integration and Continuous Delivery pipeline

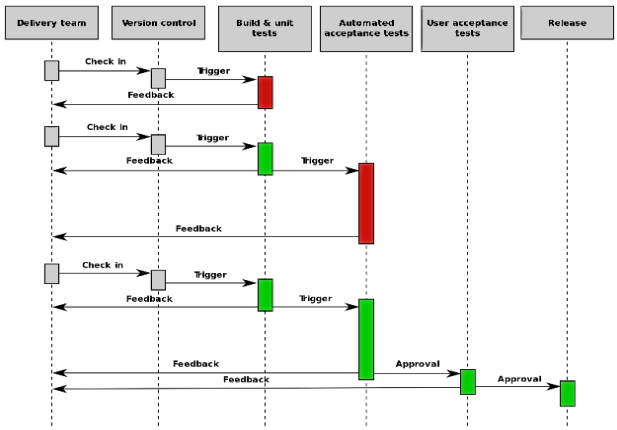

Figure 4: Feedback mechanism in CI/CD

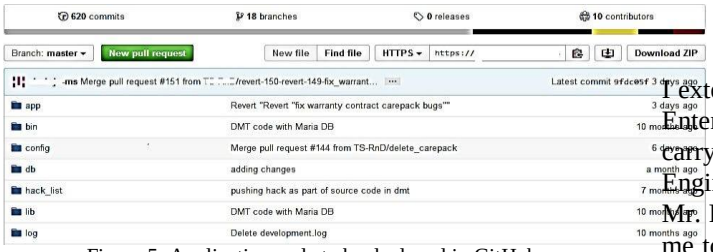

Figure 5: Application code to be deployed in GitHub

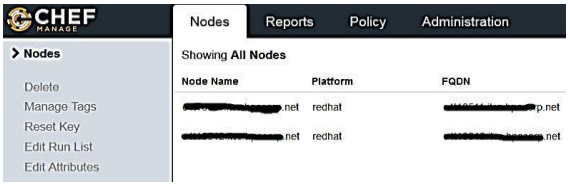

Figure 6: chef server

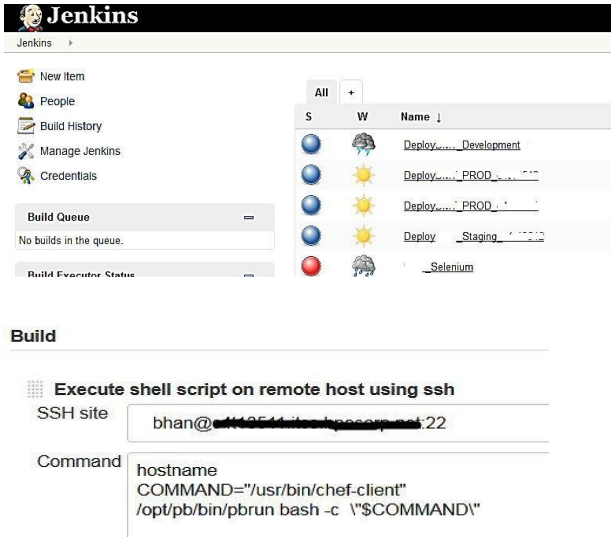

Figure 7: Jenkins job list and configurations

CONCLUSION

Management is vital to the success of every project. Good management creates processes enabling efficient delivery of software, while ensuring that risks are managed appropriately and regulatory regimes are complied with.

Finally, it is demonstrated that iterative delivery, combined with an automated process for building, deploying, testing, and releasing software exemplified in the deployment pipeline, is not only compatible with the goals of conformance and performance, but is the most effective way of achieving these goals. This process enables greater collaboration between those involved in delivering software, provides fast feedback so that bugs and unnecessary or poorly implemented features can be discovered quickly, and paves the route to reducing that vital metric, cycle time. This, in turn, means faster delivery of valuable, high-quality software, which leads to higher profitability with lower risk. Thus the goals of good governance are achieved.

ACKNOWLEDGEMENTS

I extend my sincere thanks to TSS R&D, Hewlett Packard Enterprise, Bangalore for giving me this opportunity to carry out this project. I am Grateful to Senior Software Engineering manager Mr. Aravind Desai and my mentor Mr. Ravi Shekhar Jha and other colleagues for supporting me to work on this project. I am also thankful to Head of Department Computer science, Dr. Krishna A N, SJBIT for his sincere support.

REFERENCES

[1] David Farley; Jez Humble –Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, –Published by Addison-Wesley Professional, 2010.

[2] Beck, Kent, and Cynthia Andres, "Extreme Programming Explained: Embrace Change (2nd edition)", Addison-Wesley, 2004.

[3] Mischa Taylor, Seth Vargo "Learning Chef" Publisher: O'Reilly Media, Inc. Release Date: November 2014.

[4] George Spafford, Kevin Behr, Gene Kim, –The Phoenix Project" Publisher: IT Revolution Press, Release Date: January 2013.

[5] Donald Simpson, "Extending Jenkins" Publisher: Packt Publishing, Release Date: December 2015.

[6] Ben Straub, Chris Dawson, "Building Tools with GitHub", Publisher: O'Reilly Media, Inc. Published: February 2016.

[7] Adzic, Gojko, –Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing‖, Neuri, 2009.

[8] Cohn, Mike, –Succeeding with Agile: Software Development Using Scrum‖, Addison-Wesley, 2009.

[9] Beck, Kent, and Cynthia Andres, –Extreme Programming Explained: Embrace Change (2nd edition)‖, Addison-Wesley, 2004.

[10] Duvall, Paul, Steve Matyas, and Andrew Glover, "Continuous Integration: Improving Software Quality and Reducing Risk", Addison-Wesley, 2007.

[11] ThoughtWorks, Inc., ―The Thought Works Anthology: Essays on Software Technology and Innovation‖, The Pragmatic Programmers, 2008.
[12] Clark, Mike, ―Pragmatic Project Automation: How to Build, Deploy, and Monitor Java Applications‖, The Pragmatic Programmers, 2004.

BIOGRAPHIES OF AUTHORS

Bhanupriya H, is currently studying 4th sem M.Tech (CSE) in SJBIT, Obtained Bachelor's degree under VTU. Presently working as an Intern, Hewlett Packard Enterprise, Bangalore as part of DevOps team. Has a publication in National Conference. Interested in Research in Automations.

Dr. Krishna A N is currently Professor & Head, Department of Computer Science and Engineering, SJB Institute of Technology, Bengaluru. He obtained his Bachelors and Master's degree in Computer Science and Engineering from University Visvesvaraya College of Engineering, Bangalore University, Bangalore and PhD degree from Visvesvaraya Technological University, Belagavi, Karnataka, India. He has publications in international conferences and journals. His research interests include image processing, pattern recognition and content-based image retrieval.

Ravi Shekhar Jha obtained Master's Degree in Computers and Technology from Mysore University, has work experience of 8.5 years. Presently working as DevOps Specialist, TSS R&D, Hewlett Packard Enterprise, Bangalore. He has worked extensively in different automation technologies and has a passion for automating processes as much as possible at every stage of software development lifecycle.