International Journal of Core Engineering & Management
Volume-6, Issue-07, 2020 ISSN No: 2348-9510

MANAGING AND INTEGRATING BUILD TOOL AND VERSION CONTROL SYSTEM FROM CODE TO DEPLOYMENT

Satyadeepak Bollineni
Sr. DevOps Engineer
Databricks,
Texas, USA

Abstract

In this context, integrating build tools and VCS into the workflow has become essential in modern software development processes. While automating development processes from writing code to deployment, these tools enhance collaboration, reduce errors, and speed up the software lifecycle. This paper aims to show the importance of both build tools and VCS, with great emphasis on their integration into Continuous Integration/Continuous Deployment, which becomes crucial in today's rapid development environments. Critical challenges during integration, such as merge conflicts, dependency issues, and code version management, are discussed, along with potential solutions that mitigate these problems. Bound with case studies and visual diagrams, the rather practical view is drawn of how companies like Facebook applied these tools to optimize their software pipelines. This research study discusses the challenges and best practices of integrating build tools and Version Control Systems. It conveys the automation in software deployment, which is of great significance, giving insight into the future of such technologies as expected before 2020.

Keywords: Build tools, Version control systems, Continuous Integration, Deployment pipelines

I. INTRODUCTION

Altogether, in the fast-evolving world of software, it is the management of transitions from code creation to actual deployment with sophisticated tools and mechanisms that polish up complex workflows. Build tools and Version Control Systems have changed how developers have hitherto handled, compiled, and deployed code. This fundamentally enables the automation of critical processes that were otherwise manual and time-consuming to execute. With tools like Apache Maven, Gradle, and Ant, compilation, testing, and packaging are automated. The fact that most of these steps in the workflow are automated means much reduction in human error, hence developers maintain orderliness. Version control systems maintain periodic snapshots of changes, multiple versions, and keep consistency across numerous developers, branches, and features in source code. An example would be Git and Subversion. Besides, larger software projects are scaling with growing complexity; hence, the management and integration of these tools is becoming indispensable to optimize the deployment pipeline for smooth delivery.

Performing effective management and integrating build tools with Version Control Systems are some of the major challenges in today's software development. Each of them had its own problems, including delayed deployment, building failures, and code version conflicts before

International Journal of Core Engineering & Management
Volume-6, Issue-07, 2020 ISSN No: 2348-9510

proper integration took place. In the wake of this, continuous integration/continuous deployment, in general, known as the CI/CD pipeline, adopted by companies to automate testing and deployment procedures, the integration with build tools and VCS has become highly crucial than ever before. Without their proper integration, it might be truly tough for the development

II. BACKGROUND AND LITERATURE REVIEW

1. Problem Statement

Teams to keep the code quality high, work together as expected, and, at the same time, also meet the rigid deadlines related to the project timeline. Hence, finding a proper way of incorporating these vital tools becomes highly desirable and, therefore, not a nicety but an inch of success towards any today's software undertaking.

2. Objective

It elaborates on how effective integration of build tools and VCS together amplifies the whole software development lifecycle, starting from inkling the initial code to deployment. The paper shall go on to present insight into how the deployment pipelines can be optimized, the efficiency of development augmented, and errors minimized by examining the various tools, techniques, and strategies adopted by leading companies for integrating these systems. It will also discuss best practices in vogue and found to succeed in real-life scenarios, thus giving a clear picture of how proper integration can achieve immense improvement in workflows.

3. Scope

The paper that follows provides an overview of the roles that build tools and VCS play, considers in detail the importance of each system, and discusses how they complement each other once one incorporates them into a Continuous Integration/Continuous Deployment pipeline. It discusses in full the best practices, tools, and techniques that make this happen. Real-life case studies will show, too, how they integrated and incorporated these tools into their software development. Apart from this, integration-related challenges will be outlined, including recommendations on how these challenges can be overcome, in order to show that proper tool management makes for one of the most important parts in modern software development.

4. Introduction to Build Tools

Build tools are software utilities that play an essential role in the automation of compiling, testing, and packing code into an executable format. This is one of the critical tasks for making a software product. In this regard, the primary goal of these facilities is to reduce the possibility of human error, hence increasing efficiency while ensuring that the software is built consistently using predefined standards and configurations. With large teams and complex projects at stake, a build tool's role has become indispensable in modern development environments for managing tasks, handling dependencies, and orchestrating this workflow so that developers can focus on aspects such as code quality and feature development rather than getting bogged down by the same repetitive manual tasks.

International Journal of Core Engineering & Management
Volume-6, Issue-07, 2020 ISSN No: 2348-9510

5. Popular Build Tools -pre-2020

Until 2020, it can be said that a set of different build tools emerged out of a dominant solution set because all these provided powerful automation and had been reasonably popularly adopted. Some of the most popular build tools will include the following:

Apache Maven - A widely used and acknowledged build tool known as a champion of project build, reporting, and documentation from one repository. Maven depends on a standardized POM to ease project management and, due to this, is highly used to bring order

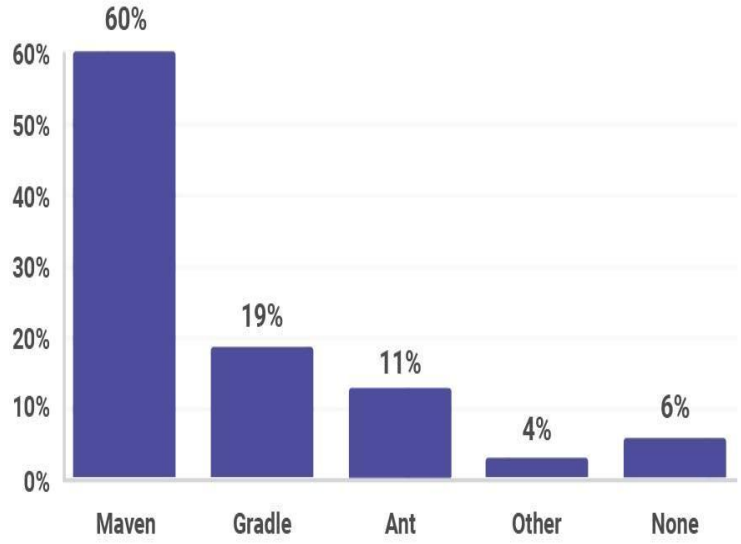

Figure 1: Maven's Dominance in build

This bar chart above shows the dominance of Maven as a build tool back in 2018, commanding a wide margin of about 60% of all usages for developers. The second runner-up, Gradle, is way behind at 19%, while Ant represents 11%. The remaining build tools fall into the small usage category of about 4%, while 6% of the developers said they did not use build tools at all. This graph reveals that 2018 Maven showed an extreme bias, meaning this tool was the favorite solution for many development teams at that time [1]. Perhaps that is because of its feature-rich set and wide support in the community.

Gradle is a very flexible tool and does multi-project builds, hence making it a well-featured dependency management system. The Groovy-based domain-specific language makes it easy for developers to automate tasks while integrating with tools like Jenkins, which makes it a favored choice for CI/CD pipelines [2].

Ant - Apache Ant is one of the oldest and most trusted build tools and is still used to automate the build process. Ant provides very flexible scripting language, which the developer uses to perform the build process according to his requirements.

III. ADVANTAGES IN THE DEPLOYMENT PROCESS

Build tools significantly reduce the time between code and deployment by automating the most meticulous tasks, guaranteeing code coherence, and managing project dependencies. In such a way, by automating these vital processes within build and deployment, developers have more time to concentrate on other, far more critical areas of the project, developing code and testing it while dropping the risk of errors caused by manual operations [3]. The automation level will extend to ensure that builds are conducted precisely the same across environments for more robust deployment.

IV. INTRODUCTION TO VERSION CONTROL SYSTEMS (VCS)

1. Definition and Purpose

Version control systems are software programs designed to assist a team in managing and tracking changes to the source code over time. These systems allow the developers to track changes within the code, go back to any point, and work together on shared projects with no possibility of work being erased or overwritten. A VCS foremost excels in contemporary software development by enabling multiple developers to work on one project simultaneously while keeping track of its history. VCS can provide a central repository where all code changes are kept, and it helps developers collaborate more effectively. It also ensures that, through the creation process, the integrity of the codebase will be maintained.

2. Types of VCS

There exist two major version control system types; both are of critical importance in the software developments done so far by 2020

3. Centralized VCs

Centralized VCSs, such as Subversion (SVN), are hosted in one location and house all their code in one single repository within a central server. All the developers check out copies of the project, modify, and commit back to that central server. This system provides easier control and management for smaller teams; however, it has less flexibility when dealing with big distributed teams that work on several features simultaneously.

International Journal of Core Engineering & Management
Volume-6, Issue-07, 2020 ISSN No: 2348-9510

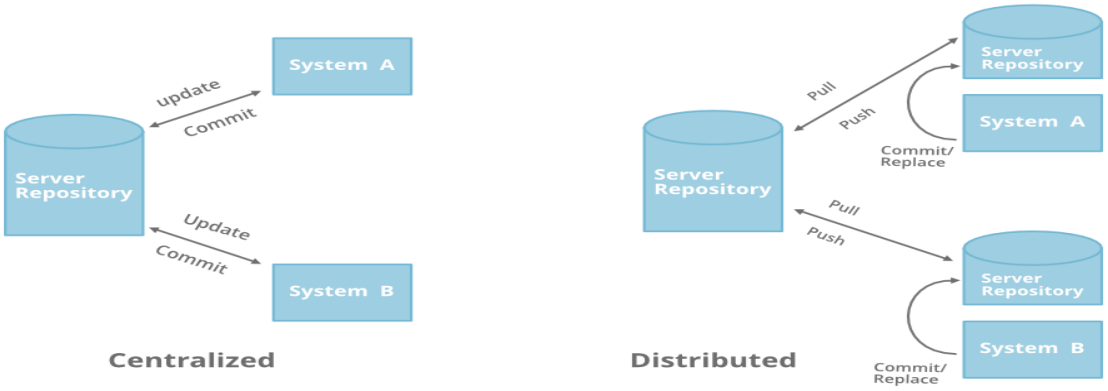

Figure 2: Comparison between Centralized and Distributed version control, systems

Figure above compares Centralized and Distributed version control systems. On the left, within the centralized model, all users commit and update their work according to that one server repository. Such an approach means that all changes flow through one central hub, which might be easier to manage but could also slow down collaboration for larger teams. In the Distributed model, each user has a copy of the repository locally. Changes can be pulled from and pushed to the server, but users can also work independently by committing modifications locally. This model further provides the added advantage of flexibility, especially for distributed teams, to work independently with less dependence on a single server [4].

4. Distributed VCS

Unlike centralized ones, Git and most other distributed version control systems create possibilities for each developer to get a full local copy of the repository [5]. That can allow the developers to make changes in their copies locally and only push their changes when they are ready to the central repository. Git has gained popularity because of its high speed, flexibility, and also for its ability to manage complex branching and merging strategies. Being distributed means Git is capable of handling huge teams involving hundreds of contributors working on different branches.

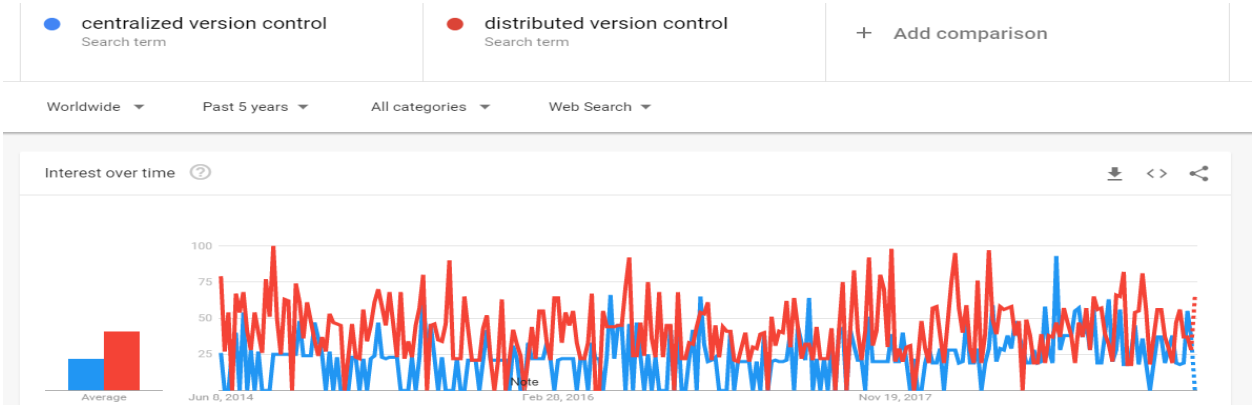

Figure 3: Interest over time for Centralized and Distributed version control systems

This graph above compares the relative interest in searches between centralized version control and distributed version control over the previous five years. This graph suggests that DVCS systems are always more interesting than CVCS systems. The red line represents the DVCS and frequently peaks above the blue line, representing the centralized counterpart. This thus indicates an ever-increasing interest in distributed version control systems like Git compared with the traditional, centralized ones such as Subversion [4]. This trend points out how the industry is moving toward flexible and scalable solutions; indeed, the distributed nature of such a system could allow several developers to work independently and then integrate the changes made by different members without several problems. While they still see use today, centralized version control systems are far behind in popularity against the rising dominance of distributed version control systems, especially considering most modern development environments that base a large portion of their workflow on heavy collaboration and massive projects.

V. VERSION CONTROL IN DEPLOYMENT PIPELINES

Version control systems tend to give many privileges in deployment pipelines. VCS enables developers to track various types of changes across code efficiently. Moreover, this allows multiple team members to collaborate on the codebase while the code is kept organized and correctly versioned. This integration of VCS within CI/CD pipelines helps teams ensure that only tested, clean code is deployed into production environments, which reduces the risk of any failures in deployment [6].

Moreover, VCS acts like a safety net in that developers can quickly revert to earlier code versions in case of any hitch during deployment. This ensures that any problem is resolved as expedited. The branching and merging capabilities of the VCS also allow the simultaneous development of different features with no conflicts.

VI. INTEGRATION OF BUILD TOOLS AND VERSION CONTROL SYSTEMS

1. CI/CD Pipelines

Continuous Integration/Continuous Deployment has become an indispensable part of software development, where code testing, building, and deployment are automated for new changes to smoothly reach the main branch. Version control and building tools form a necessary basis for any CI/CD pipeline, which will automate key activities: compilation of code, running tests, or versioning code [7]. Tools like Jenkins, Travis CI, and CircleCI allow project integration by automatically running the tests and building after every push to a repository. It will save lots of manual effort and help detect and correct errors much earlier in the development phase.

International Journal of Core Engineering & Management
Volume-6, Issue-07, 2020 ISSN No: 2348-9510

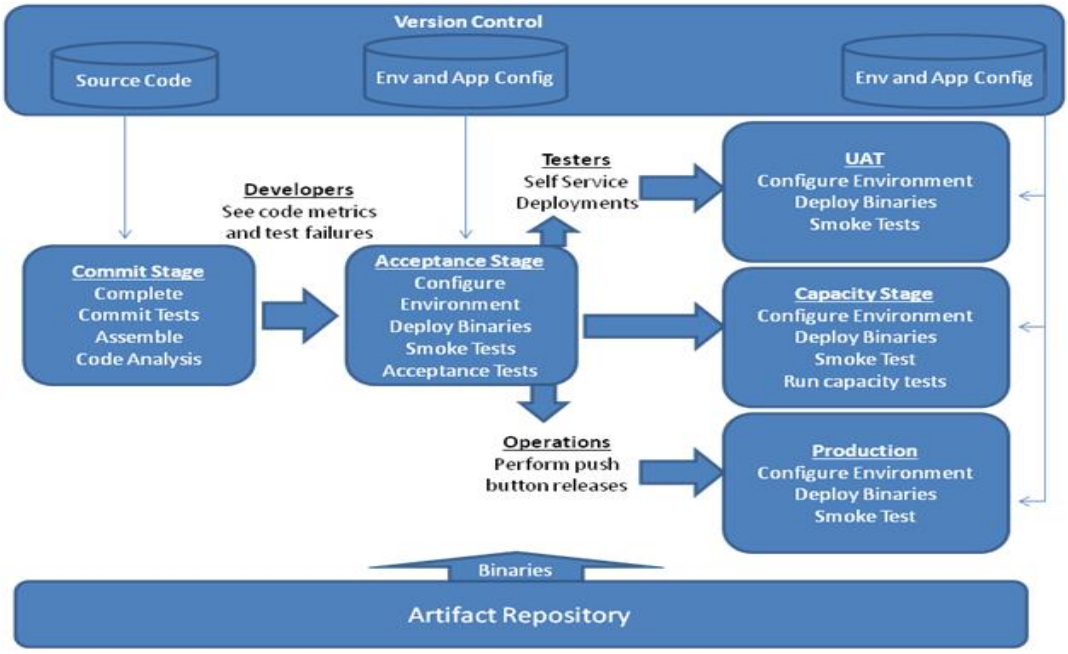

Figure 4: Software Deployment Pipeline

Above is a neat software development pipeline diagram, which effectively incorporates Continuous Integration (CI), Continuous Deployment (CD), and Version Control Systems (VCS). In this flow, the developers generally work with the source code and environment/app configuration stored in version control. Then comes the commit stage where the code is tested, assembled, and analyzed and passes through acceptance, capacity, and production [8]. Each step will include the deployment of binaries, provisions for environment configuration, running tests, and analysis of results. Continuous feedback loops will provide developers with code metrics and failures in real-time. Testers perform self-service deployments. Binaries generated during that pipeline are stored in the artifact repository because these need to be traceable. Operations execute push-button releases into production, ensuring the application's automated deployment proceeds seamlessly. This is made possible by integrating VCS into CI/CD pipelines, continuous testing, rapid feedback, and reliable computerized deployments that are much more effective and less error-prone.

2. Around Integration Issues

While integrating build tools and VCS into the CI/CD pipeline may improve development efficiency, performing these tasks in this pipeline is challenging. The most common problems developers experience are merge conflicts when changes from multiple developers clash to cause errors during the integration process [9]. There are also dependency problems when some parts of the codebase depend on incompatible versions of libraries or tools, which can lead to build breaks and slow down deployment. Other common challenges that occur during deployment include misconfigurations of environments, untested changes, and build breaks that can disturb the process of deployment and reduce the reliability of the CI/CD pipeline [12].

International Journal of Core Engineering & Management
Volume-6, Issue-07, 2020 ISSN No: 2348-9510

3. Best Practices

These challenges have been overcome by bringing in a set of best practices that ensure the smooth integration of build tools and version control systems into the CI/CD pipelines. These are:

1) Frequency of Commits:

It will reduce the merge conflicts arising out of complexity, as frequent committing compels the developer to commit the code in small pieces. It becomes easier to handle and integrate minor, incremental updates rather than massive or significant code-level changes. In such a case, the chances for potential conflicts lessen, while tracing could be more accessible.

2) Automated Testing:

Full-featured automated testing suites are set up to run with every commit or pull request, ensuring that code will remain quality-minded throughout the development process. They find errors early before they can be merged into the main branch, significantly reducing the possibility of failed builds and broken deployments.

3) ***Branching Strategy*****:**

Utilize clearly defined branching strategies, such as GitFlow, which ensure proper segregation between the development, testing, and production environments. Developers could work in separate feature-related, bug-fix, or experimental branches that deposit into the master only when the code has been tested and cleared [11]. This way, incomplete or unstable code could not enter the production pipeline.

4) Continuous Feedback Loop:

Setting up the constant feedback loop will help developers contribute to the CI/CD pipeline for fast detection and resolution of issues. Since the notifications are provided immediately in case a build fails, a test fails, or there are problems with a merge conflict, developers can take immediate action.

5) Application Environment Consistency:

For builds, tests, staging, and production, maintain consistency to avoid build and deployment failures. Containerization may be considered in attempting to keep applications consistent in behavior throughout the entire life cycle of a CI/CD pipeline. This will enable organizations to make their development processes more effective, reduce possible failures in deployment, and deliver new features and fixes to end-users most effectively and efficiently.

VII. CASE STUDIES OF INTEGRATION

Case Study 1: Facebook

For fast development and deployment cycles, the highest praise goes to Facebook. With integrated tools like Git for version control and Jenkins as an automation tool, Facebook has maintained an efficient CI/CD pipeline. Hence, it can deploy code thousands of times a day [10]. Versioning control with Git allows developers to work on more than one feature concurrently, while Jenkins automates continuous integration to test and build before deployment. In this respect, Facebook

has minimized downtown, smoothened its development workflows, and significantly reduced the chances of failure in deployment. It also lets Facebook include a variety of tools, providing the company with the ability to hasten feature releases and updates without creating a dent in their customers' experience.

Case Study 2: Health Care Industry

In the medical industry, CI/CD pipelines and modern DevOps practices play an important role in compliance with HIPAA regulations and patient data security. Healthcare organizations use version control with Git, Jenkins for continuous integration, and Gradle for continuous deployment. Early challenges they normally face include dependency management and alignment of development and production environments. However, through automated testing and standardized environments, they come out to improve efficiency and reduce errors. Via the CI/CD pipeline, faster release cycles and continuous monitoring are allowed-which in turn makes sure updates that block strict security and regulatory standards are met as part of the procedure. This leads to a higher quality of software, expedited deployment times, and protection of patient data that will, in turn, help an organization maintain agility within the strict healthcare regulations.

VIII. CONCLUSION

In conclusion, the integration between building tools and source control systems plays a very vital role in ensuring deployed software is both efficient and reliable. Because they automate the process of building through code version management, these tools assist developers in keeping maintained quality code with minimal errors and enabling swift deployment.

These benefits accrue further when one uses continuous integration/ continuous deployment pipelines because automation phases of testing and deployment will enable delivering updates and features quickly with minimal disruption.

REFERENCES

1. Snyk,"10 Maven Security Best Practices1", Sep. 26, 2018.
2. Rafel Leszko, "Continuous Delivery with Docker and Jenkins Delivering software at scale, Aug 2017.
3. 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
4. M. Lescisin, Q. H. Mahmoud, and A. Cioraca, "Design and Implementation of SFCI: A Tool for Security Focused Continuous Integration," Computers, vol. 8, no. 4, p. 80, Nov. 2019.
5. M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: assurance, security, and flexibility," Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Aug. 2017.
6. "Memory Leaks and GDI-VBForums," Vbforums.com, 2017.
7. G. Wright, "Continuous Integration (CI)," Medium, Jul. 13, 2018.
8. Snehamayee, "Deployment Pipeline," Welcome to World of Agile, Mar. 31, 2018

9. M. DeKrey, "A scalable Git branching model - Matthew DeKrey - Medium," Medium, Aug. 11, 2017.
10. S. A. I. B. S. Arachchi and I. Perera, "Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management," IEEE Xplore, May 01, 2018.
11. H. Anzt, Y.-C. Chen, T. Cojean, J. Dongarra, G. Flegar, P. Nayak, E. S. Quintana-Ortí, Y. M. Tsai, and W. Wang, "Towards continuous benchmarking: An automated performance evaluation framework for high-performance software," in Proceedings of the Platform for Advanced Scientific Computing Conference (PASC '19), Zurich, Switzerland, Jun. 2019, pp. 1-11.
12. H. Myrbakken and R. Colomo-Palacios, "DevSecOps: A Multivocal Literature Review," Communications in Computer and Information Science, pp. 17–29, 2017.
13. M. Rao, "Common security challenges in CI/CD workflows," Synopsys.com, Jul. 10, 2018.