



Ministry of Higher Education and Scientific Research

Tunis El-Manar University

National Engineering School of Tunis

Department of Information Technologies and Communication

End-of-Year project 2

---

# Set up a reliable CI/CD pipeline based on continuous monitoring

---

*Elaborated by:*

Aya CHERIGUI

2<sup>st</sup> year in Telecommunications Engineering Studies

*supervisor:*

Professor Mohamed Escheikh

University Year: 2021/2022

## Acknowledgement

This work would not have been possible without the help of my dearest mentor Mr. **Mohamed ESCHEIKH**, professor and senior lecturer at ENIT. I would sincerely like to thank him for his inestimable guidance, his availability and especially his judicious advice which contributed to feed my reflection and to conduct this work well.

# Contents

<b>List of Figures</b>	<b>4</b>
<b>Abstract</b>	<b>7</b>
<b>Introduction</b>	<b>8</b>
<b>1 The Evolution of Software Delivery</b>	<b>9</b>
1.1 Introduction . . . . .	9
1.2 Traditional Software Development . . . . .	9
1.3 Modern Software Development . . . . .	11
1.3.1 The Agile Manifesto . . . . .	11
1.3.2 Deployment on The Cloud . . . . .	12
1.3.3 The DevOps Approach . . . . .	13
1.4 Conclusion . . . . .	16
<b>2 Continuous Integration, Continuous Delivery and Deployment</b>	<b>17</b>
2.1 Introduction . . . . .	17
2.2 Pre-automation Delivery Process . . . . .	17
2.3 Continuous integration . . . . .	19
2.4 Continuous Delivery and Deployment . . . . .	21
2.5 Continuous monitoring and observability in CI/CD . . . . .	23
2.6 Conclusion . . . . .	24
<b>3 Use Case: Implementing a CI/CD pipeline</b>	<b>25</b>
3.1 Introduction . . . . .	25
3.2 Git: a VCS tool . . . . .	25
3.3 Jenkins: a CI tool . . . . .	26
3.3.1 Jenkins Architecture . . . . .	27
3.3.2 Jenkins Installation . . . . .	28
3.3.3 Jenkins Plugins . . . . .	28
3.3.4 Jenkins Configuration . . . . .	28

3.4	Set up the repository . . . . .	28
3.5	A Jenkins Job Build . . . . .	28
3.6	Conclusion . . . . .	30
<b>General Conclusion</b>		<b>31</b>
<b>Bibliography</b>		<b>32</b>
<b>Appendix</b>		
<b>A Jenkins Dashboard</b>		
<b>B Jenkins Configurations</b>		
<b>C Jenkins Job Build</b>		

# List of Figures

1.1	Software Development Life Cycle [1]	10
1.2	Waterfall versus Agile [2]	12
1.3	DevOps life cycle [3]	14
2.1	Pre-automation Delivey Cycle [4]	18
2.2	Continuous Integration [5]	20
2.3	Continuous delivery Versus Continuous deployment	22
2.4	The CI/CD pipeline	23
3.1	Jenkins Architecture [6]	27
3.2	Console output	29
A.1	Jenkins dashboard	
B.1	System Configuration screen	
C.1	System Configuration screen	
C.2	Jenkins Project Choices	
C.3	Source Code Management integration	
C.4	Build Triggers	
C.5	Git Webhook Settings	

# Acronyms

<b>AWS</b>	Amazon Web Services
<b>CD</b>	Continuous Delivery/Deployment
<b>CI</b>	Continuous Integration
<b>CM</b>	Continuous Management
<b>Dev</b>	Development
<b>FTM</b>	Faster Time to Market
<b>HTTP</b>	Hypertext Transfer Protocol
<b>IaC</b>	Infrastructure as Code
<b>IT</b>	Information Technology
<b>KPI</b>	key performance indicator
<b>MTTF</b>	Mean Time To Failure
<b>MTTR</b>	Mean Time To Repair
<b>Ops</b>	Operations
<b>QA</b>	Quality Assurance
<b>SCM</b>	Source Code Management
<b>SDLC</b>	Software Development Life Cycle
<b>SSH</b>	Secure Shell
<b>UAT</b>	User Acceptance Testing
<b>URL</b>	Uniform Resource Locator

**VCS** Version Control System

# Abstract

In this project, we will set up a CI/CD pipeline for a simple application using the CI tool, Jenkins. A pipeline helps us automate steps in the software delivery process, such as introducing automatic builds. We will use Jenkins, a service that builds, tests, and deploys the code every time there is a code change, based on the release process models defined. As part of our setup, we will plug other DevOps tools into Jenkins to complete the software delivery pipeline. The other tools which are being used are GitHub and JUnit. GitHub acts as source code repository and JUnit acts as a unit testing framework for the Java programming language. The core concept of CI and CD are presented throughout the project.

**Key words:** CI/CD, Jenkins, DevOps, automatic, Github, JUnit, Java



# General Introduction

Digitization offers many benefits: speed, efficiency and agility. Its goal is to speed up workflow processes, give employees more time to focus on important work and give them the ability to be agile in order to meet new expectations, requirements and trends.

In the software industry, while Agile focuses on communication between customers and developers, allowing them to understand each other and work smoothly to achieve goals, DevOps focuses on bridging the gap between the development team and the operations team.

The concept of DevOps is based on the collaboration and sharing of responsibilities between the development team and the operations team in order to improve the quality of the product by solving problems together. This approach was introduced in software development to extend the principles of agile software development. It improves the efficiency sought by IT and development teams. It merges and prioritizes development and business requirements to solve IT problems and to agree on the processes and products to be automated during software development and deployment. This automated process is known as the continuous integration and delivery (CI/CD) pipeline.

CI/CD is a well-known DevOps practice for ensuring rapid delivery of new features and allows development teams to deliver code changes consistently into production. Continuous integration (CI) allows developers to automatically integrate new code into a shared repository and check for errors at the same time, while continuous deployment allows for frequent deployment of software into the production environment when checks are successful. Continuous Delivery (CD) helps teams keep software in a state of readiness for customer use and reduce release time to get quick feedback from users for improvements.

In the first chapter, we will outline the evolution of software development methodologies over time, from the very first waterfall model to the DevOps model. In the second chapter, we will discuss the CI/CD pipeline and its role in the modern software development life cycle. In the third chapter, we present showcase a case study, an example to illustrate the advantages of a CI/CD tool called Jenkins.

# Chapter 1

## The Evolution of Software Delivery

### 1.1 Introduction

Software development has not always worked the way it does today. Not so long ago, the development processes were very different, as was the technology available for software engineering. To understand the importance of continuous integration/continuous deployment (CI/CD) pipelines, it's essential to take a step back and look at how it all began [4].

In this chapter, we will overview software development in its early days, before automation came into play. In a second time, we will introduce the agile software development methodology and the impacts of its practices, with a primary focus on the **DevOps** approach, the DevOps lifecycle its key components, and best practices.

### 1.2 Traditional Software Development

Software development follows a flow, starting with identifying new features, planning, developing, committing the source code changes, running builds and tests (unit, integration, functional, acceptance, etc.), and deploying to production (figure 1.1) [7].

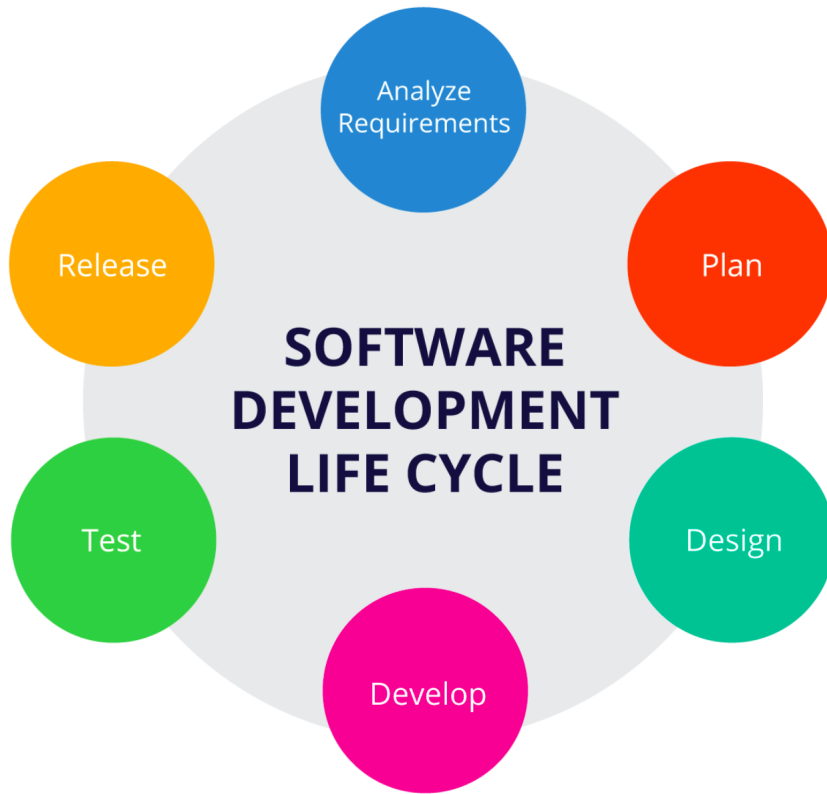


Figure 1.1: Software Development Life Cycle [1]

The waterfall is the oldest and simplest of the structured Software Development Life Cycle (SDLC) methodologies; it's based on completing one phase, then move on to the next. There is no turning back. Each phase builds on the output of the previous phase and has its own project plan, which is why it is called a waterfall methodology. Once the software is fully developed, the operation team deploys the software according to the specifications from the planning phase.

The waterfall method is easy to understand and adopt. Yet, early delays can disrupt the entire project schedule. And because there is little time for revisions once a stage is complete, problems cannot be solved until maintenance stage is reached. The waterfall model does not work well if flexibility is required or if the project is long-term and long-lasting.

## 1.3 Modern Software Development

Software is increasingly important to the functioning of the world as we know it. As a result, the complexity of that software is also increasing and new types of bugs and security holes are popping up. Therefore, the way software is developed hasn't stopped evolving and changing to best meet the growing needs of the consumers, resulting in competition within the software industry.

In this section, we present the major changes that the software delivery world has encountered and the challenges it has overcome overtime.

### 1.3.1 The Agile Manifesto

The agile manifesto was established to create a lightweight set of values and principles against heavyweight software development processes and methodologies such as the unified process and waterfall development [7].

With a traditional “waterfall” software development approach, developers could work independently for a very long time. They would have no idea about how many issues they would encounter during the integration phase. To solve this issue, the “Agile” software development model was introduced.

Agile changed the way software development teams worked. One of the key principles was to frequently deliver working software in short time frames, with an emphasis on the desire for small batch sizes and incremental releases rather than large waterfall releases which took months and even years.

Delivering software frequently meant producing stable code for each and every incremental release.

Waterfall software development involves one main phase, while agile development relied on smaller cycles based on feedback from the previous iteration (Figure 1.2) [7].

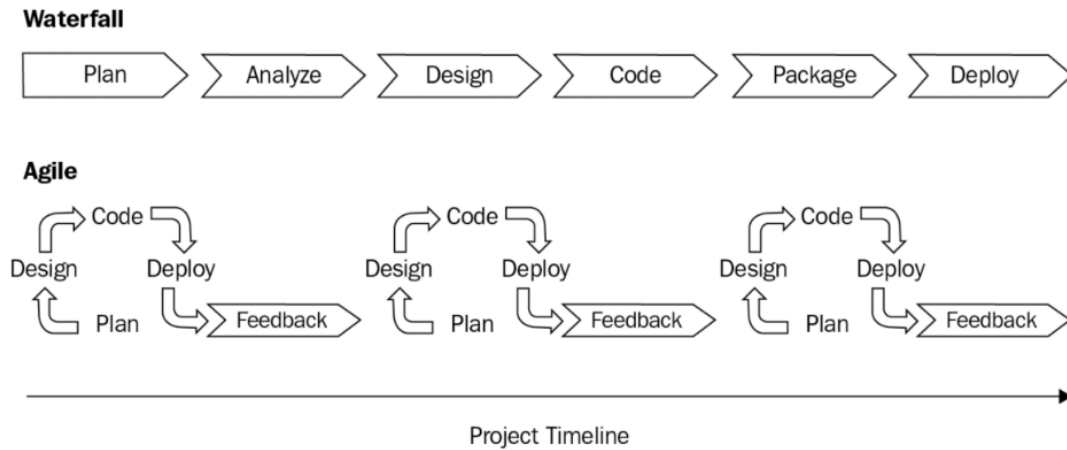


Figure 1.2: Waterfall versus Agile [2]

Agile is an empowering process. When used successfully, project teams are capable of self-organization and cross-functionality. However, it requires endless collaboration between different development teams that have been working on different modules of the software simultaneously. It was quite a challenge to integrate changes from different developers working amongst different teams. This led to software teams looking for better, more reliable approaches.

### 1.3.2 Deployment on The Cloud

The cloud has drastically changed the way applications were created and maintained. Until then, most software development businesses or web companies had their own servers and an operations team to maintain them. With the advent of on-demand cloud services, all that has changed. It is now possible to create a new environment and run it directly on a third-party infrastructure. This new approach has saved the time spent managing actual hardware and made it easier to create reusable environments [4].

The cloud made deployment easier than ever. A software developer could run a virtual machine with the necessary software and runtime engines, then run a batch of unit tests to ensure it was the specific server. It's also possible to create an environment where customers can see changes to the application at the end of each iteration, helping them approve those new features or provide feedback on a requested improvement.

Server environments were easier to start, faster to scale, and less expensive than

actual hardware. The migration to cloud-based software development has made it easier to automate many processes related to software deployment practices. Using a command line tool, it is now possible to start a new test and production environment, create a new database and even eliminate unnecessary servers.

More and more companies are on the web, and the competition to release new features or implement the same features as your competitors has become a challenge. It became not reasonable to deploy every few months. If a competitor released a new feature, your product also had to implement it as quickly as possible or risk losing market share. If a bug fix was delayed, it also meant a potentially large loss of revenue. These changes have been at the heart of a revolution in the way software is built. Until now, software development involved teams of software engineers overseeing the design of new features, fixing bugs and preparing future releases. On the other side, a group of system administrators ensured that the entire infrastructure was working properly and that no bugs were introduced into the system.

Although these two teams had the same goal of making the software work better, they found themselves in conflict because of the nature of their work. The developers were under a lot of pressure to get releases out faster, but they were likely to introduce bugs or require software updates to the servers with each release. Sysadmins in turn were also under pressure to keep the environment stable and pushed changes to avoid upsetting the delicate balance of existing systems. This dichotomy gave rise to a new philosophy in the technology industry known as DevOps [8].

### **1.3.3 The DevOps Approach**

DevOps is a cultural philosophy that emphasize the collaboration between the development (Dev) team and the operations (Ops) team. It is a combination of practices and tools that automates and integrates processes between the two teams. Devops emphasizes team accountability, communication and collaboration between teams, and technology automation.

The DevOps lifecycle improves development processes from start to finish and engages the organization in continuous development, resulting in faster deliveries.

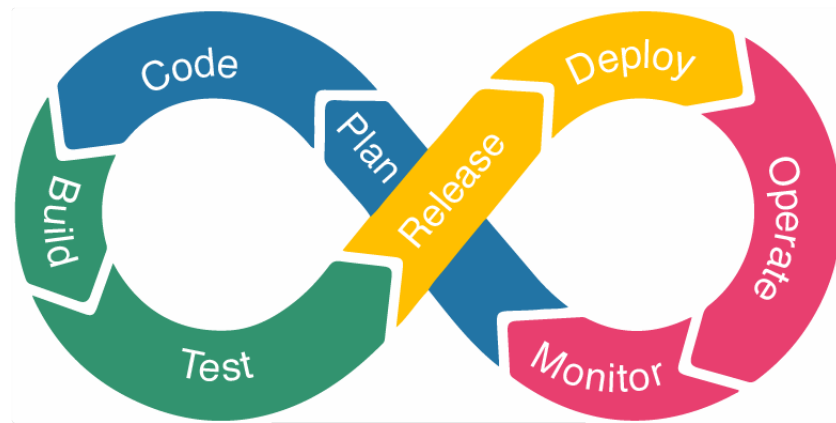


Figure 1.3: DevOps life cycle [3]

DevOps mainly consists of the following practices:

### Continuous Development

This practice covers the planning and coding phases of the product. It helps the DevOps team accelerate the overall software development process. This phase in the DevOps lifecycle is in charge of source code maintenance. There are a variety of tools for source code maintenance including JIRA, Git, Mercurial and Subversion. There are also various tools for packaging code into executable files, such as Ant, Gradle and Maven. These executable files are then passed on to the next stage of the DevOps lifecycle.

### Continuous Integration

Continuous integration focuses primarily on the automated build and testing of each change that developers make to the version control system. This step combines configuration management (CM) tools with other testing and development tools to determine how ready the code under development is for production. It involves rapid feedback between testing and development to quickly detect and resolve code issues. At the same time, customers also provide information to be incorporated to add new features to the application. Most changes are made to the source code during this phase.

### Continuous Testing

This practice incorporates automated, pre-scheduled, continuous code testing as the application code is written or updated. These tests can speed up the release

of code to production.

Automation tools, such as JUnit, Selenium, and TestNG, are used for continuous testing, which allows the QA team to analyze multiple code bases simultaneously. This makes sure that there are no deficiencies in the functionality of the developed software.

### **Continuous Delivery**

Continuous Delivery automates the delivery of code changes, after testing, to a pre-production or staging environment. A staff member is then in charge to decide whether such code changes are ready to move into the production environment.

### **Continuous Deployment**

In Devops, developers receive fast and constant feedback on their work, allowing them to quickly and independently implement, integrate, and validate their code, and ensure that the code is deployed to the production environment. They do this by continuously recording small changes in the version control repository, performing automated and exploratory testing on those changes, and deploying them to production. This gives us a high degree of confidence that our changes will work as expected in production and that any problems can be quickly detected and corrected [7].

A wide variety of tools have been designed to provide the functionality of the deployment pipeline, many of them are open source, for example, Jenkins, ThoughtWorks Go, Concourse, Bamboo, Gitlab CI, etc...

### **Continuous Monitoring**

This practice involves constant monitoring of both the code in operation and the underlying infrastructure that supports it. A feedback loop that reports on bugs or gray areas in the software then returns to development. Continuous monitoring is an operational phase where the goal is to enhance the overall efficiency of the software application as well as its performance.

### **Infrastructure As A Code**

Infrastructure as Code (IaC) is the process of managing and provisioning IT data centers through configuration files rather than physical hardware configuration and interactive configuration tools. IaC can be used during various DevOps phases to automate the provisioning of the infrastructure needed to release software. Developers add infrastructure "code" from their existing development tools. For instance, developers can create on-demand storage volume from Docker, Kubernetes



or OpenShift. This practice also allows operations teams to monitor environment configurations, track any changes and facilitate rollback of configurations.

## 1.4 Conclusion

DevOps is fundamentally changing how dev and ops are done today. It's not only about speeding up software development, but it's about improving the quality of software. DevOps brings a new mindset, agile practices, and smart tools that collectively achieve this goal.

## Chapter 2

# Continuous Integration, Continuous Delivery and Deployment

### 2.1 Introduction

By using automation, it's possible to build more robust software and release it faster. With containers and integration platforms, it's also easier to create applications that can be put into production with minimal impact on larger systems. These automation processes are known as CI/CD, defined as three distinct steps that make up a larger pipeline. These steps are continuous integration, continuous delivery, and continuous deployment [4].

In this chapter, we demystify continuous integration, continuous deployment, and continuous delivery by describing how companies traditionally deliver software and explains the idea of improving it using the continuous delivery approach. Finally, we will describe the principles of monitoring and observability, how they are related and how automation can streamline the overall deployment process.

### 2.2 Pre-automation Delivery Process

Any delivery process starts with the requirements defined by a customer and ends with the release to production. Traditionally, it is presented as shown in the figure 2.1.

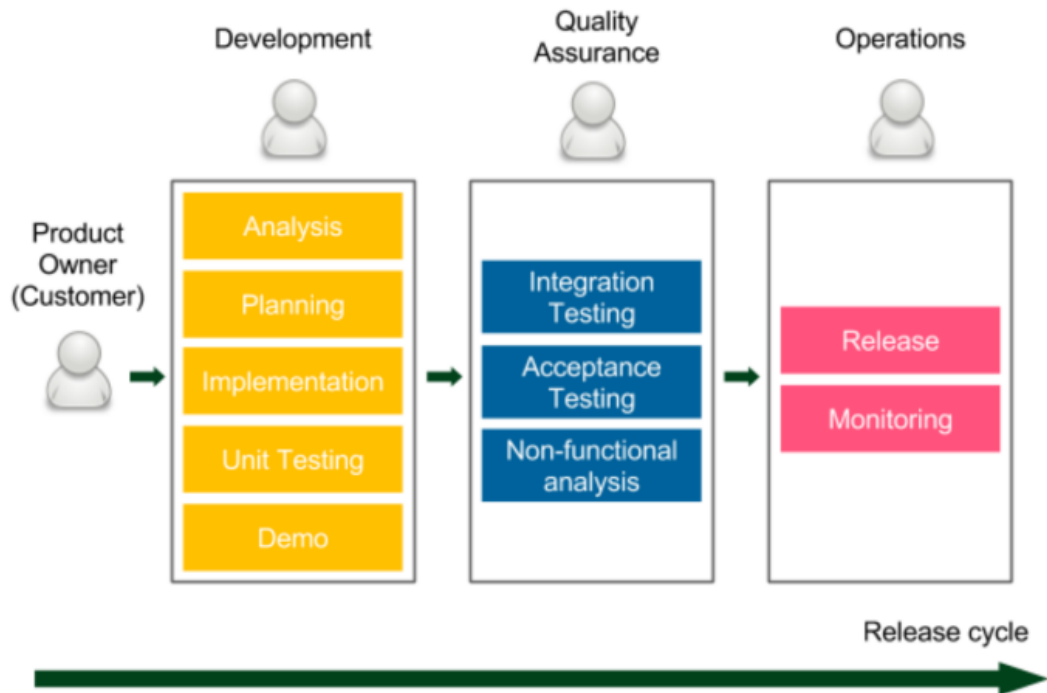


Figure 2.1: Pre-automation Delivey Cycle [4]

The delivery cycle starts with the requirements provided by the product owner, who represents the customer (stakeholders). Then there are three phases, during which the work passes between different teams:

**Development:** Developers work on the product using agile techniques (Scrum or Kanban) to increase the velocity of development and improve communication with the customer. Demo sessions are organized to get feedback from the customer. Once the implementation is complete, the code is sent to the quality assurance team.

**Quality Assurance:** This phase is called User Acceptance Testing (UAT) and it requires a code freeze on the trunk code base, so that no new development could break the test. The QA team performs a suit of Integration Testing, Acceptance testing, and Non-functional Testing (performance, recovery, security, and so on). Any bug that is detected goes back to the development team. Once the UAT phase is complete, the QA team approves the features planned for the next release.

**Operations:** The last phase, usually the shortest one, means passing the code to the operations team, so that they can perform the release and monitor the production. If anything goes wrong, they contact developers to helps with the production systems.

## 2.3 Continuous integration

Continuous integration, also known as CI, is a foundation of modern software development that focuses on the early stages of a software development pipeline where code is built and undergoes initial testing.

CI enables and triggers a series of iterative process improvements, from a simple automated scheduled build to continuous production delivery.

In the days of waterfall projects and Gantt charts, before the advent of CI practices, the development team's time and energy were regularly depleted in the period leading up to a release by what was called the **integration phase**. During this phase, changes made to the code by individual developers or small teams were assembled in pieces and turned into a working product.

This was a challenging job, sometimes involving the integration of months of conflicting changes. It was very difficult to anticipate the kinds of problems that might arise, and even more difficult to solve them because it might involve reworking code that had been written weeks or months before [9].

This tedious process was fraught with risk and danger, and often resulted in long delivery delays, unexpected costs, and, as a result, unhappy customers.

Continuous integration emerged to solve these problems.

CI uses a variety of tools and automation techniques to create builds and guide them through initial tests, such as unit tests, as well as more comprehensive integration tests. It is a process that monitors changes to the version control system.

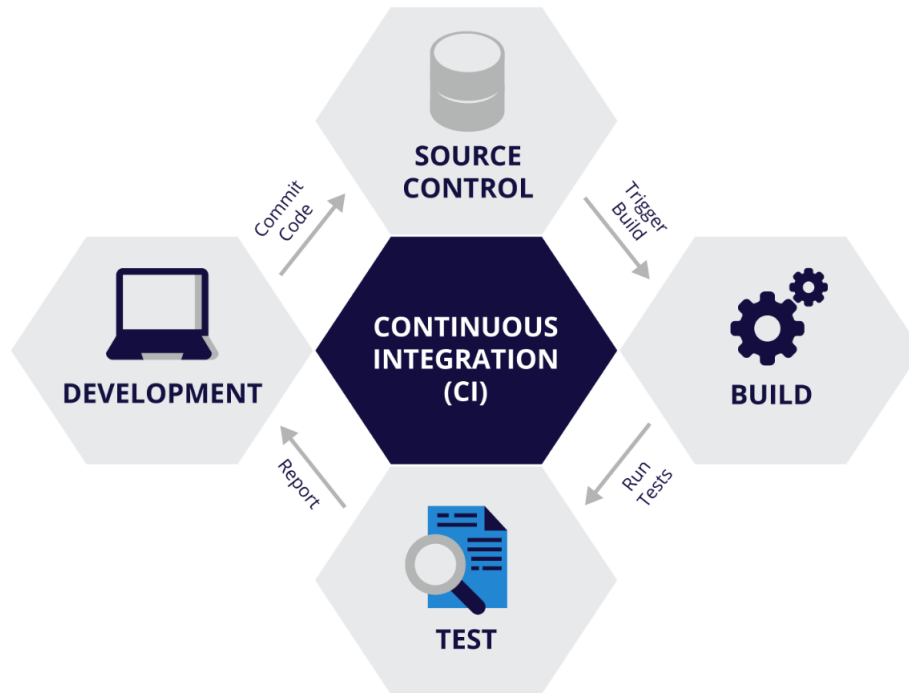


Figure 2.2: Continuous Integration [5]

In the figure 2.2, we see the CI life cycle which is triggered by committing code to source control and ends with successful testing. As soon as a change is detected in the version control system, this tool automatically compiles, builds and tests the application. The change can be a push to a remote repository or pull into a local repository, or even just a commit.

If something goes wrong, the tool informs the developers immediately so that they can fix the problem [9].

Initially, this step consisted mainly of running a series of unit tests, but it can now include other processes that make the integration reliable and effective such as installing all the project dependencies to validate the integrated code and checking for potential vulnerabilities.

The purpose of unit testing is to validate the output of a single, stand-alone function or class. These tests are typically fast and can be run on a large code base in seconds. Once all the code is valid, the CI process can merge the code into a branch for potential release to a test environment.

Continuous integration can do much more than that, it can help the developer

keep an eye on the overall health of the code base, automatically monitoring code quality and coverage metrics, and helping you reduce maintenance costs. Combined with automated end-to-end acceptance testing performed by the QA team, continuous integration can serve as a communication tool, publishing a clear picture of the current state of development efforts. Finally, it can simplify and accelerate delivery by helping developers automate the deployment process, enabling automatic deployment of the latest version of your application.

In essence, continuous integration aims to reduce risk by providing fast and detailed feedback. More importantly, it is designed to help identify and resolve integration and regression issues faster, resulting in smoother delivery and fewer bugs. The limited nature of each iteration means that bugs are identified, located, reported, and fixed with relative ease.

By providing better visibility to technical and non-technical team members on the status of the project, continuous integration can open and facilitate communication channels between team members and encourage collaborative problem solving and process improvement. And by automating the development process, continuous integration helps you get your software into the hands of testers and end-users faster, more reliably, and with less effort.

## 2.4 Continuous Delivery and Deployment

Continuous delivery is an extension of continuous integration that allows for advanced testing, including functional and user acceptance testing, configuration, and load testing. The testing is done in small incremental iterations to quickly and cost-effectively identify and resolve any issues that are revealed during testing, as compared to traditional software development approaches. Such testing validates that the build meets the requirements and is ready for use in a production environment.

Continuous Delivery triggers a final human intervention and then a push to deployment. It may also be possible to automatically deploy the build, a step called **Continuous Deployment** (see Figure 2.3). Instead of manually deploying the software into production, continuous deployment makes it possible to do that automatically.

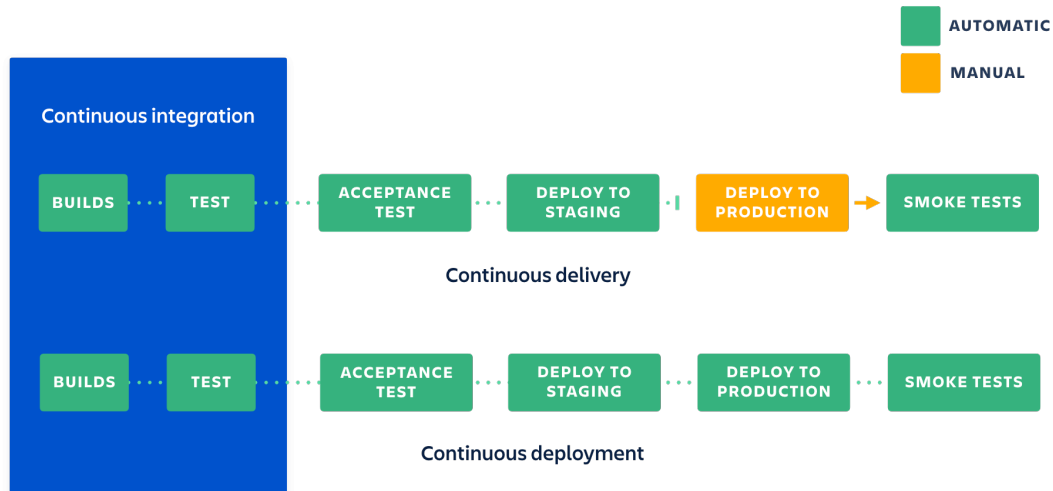


Figure 2.3: Continuous delivery Versus Continuous deployment

Indeed, if we take the automation of the deployment process to its ultimate logical conclusion, it is possible to put every build that passes the necessary automated tests into production. The practice of automatically deploying each successful build to production is generally known as continuous deployment.

A pure Continuous Deployment approach, however, is not always appropriate for everyone. For instance, many users would not appreciate a new release dropping on them several times a week, and prefer a more predictable and transparent release cycle [10].

The notion of Continuous Delivery is a slight variation on the idea of continuous deployment that takes these considerations into account. With continuous delivery, any successful build that has passed all automated testing and quality checks can be deployed to production via an automated process and be available to the end-user within minutes.

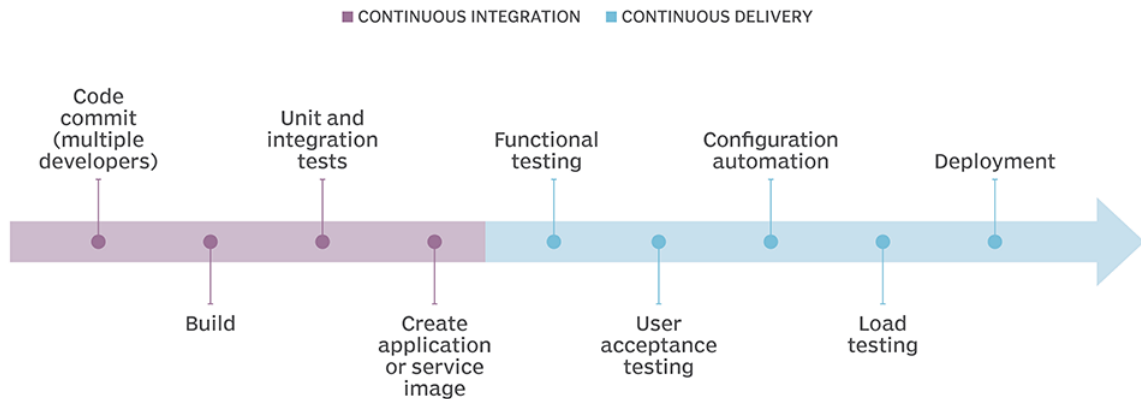


Figure 2.4: The CI/CD pipeline

Continuous integration, delivery and deployment are collectively referred to as continuous software development. the key performance indicator (KPI) of CI/CD is the deployment Time and frequency, the change Lead Time, the change Failure Rate, the mean time to repair (MTTR) vs. the mean time to failure (MTTF).

## 2.5 Continuous monitoring and observability in CI/CD

Despite obvious business advantages, a rapid release approach combined with continuous change processes resulting from the CI/CD pipeline will in the long run generate new challenges. Deploying a CI/CD pipeline is only half the work. To complete the deployment, continuous monitoring and observability (CM) must be implemented. The entire process must be carefully reviewed and monitored. Monitoring relies on collecting predefined sets of measurements or logs, while observability relies on exploring undefined properties and patterns.

Monitoring and observability solutions are designed to provide leading indicators of service degradation, detect bugs and unauthorized activity, help debug them, identify long-term trends for capacity planning and business objectives, and expose unexpected side effects of changes or added functionality. Implementing CM to the CI/CD means adopting the following practices:

**Long-term trend analysis:** This involves assessing the number of builds to be performed on a daily basis and evaluating builds performed several months ago.



Also, deciding how to scale the infrastructure.

**Vulnerability scan:** This checks if the code introduces critical software flaws and security vulnerabilities such as memory leaks, uninitialized variables, array bounds and check how fast they are detected and how fast they are fixed.

**Over-time Comparison:** To compare the pace of the deployment with previous ones.

**Alerting:** To verify if something is broken, or might be broken soon, and check whether the test pipeline was successful. If not a roll back the last deployment is needed.

## 2.6 Conclusion

Automation is the idea behind implementing CI/CD. The processes help ensure minimal sign-offs and manual authentications that otherwise hurt the lead time. It also ensures faster time to market (FTM) and helps maintain shipping cadence.

# Chapter 3

## Use Case: Implementing a CI/CD pipeline

### 3.1 Introduction

In this chapter, we will introduce the Jenkins tool, its architecture and its main features. then I will setup a simple Jenkins automated build job. Finally, I will show a concrete example of a CI/CD pipeline. But prior to that, I will cover the topic of source code management with Git.

### 3.2 Git: a VCS tool

Version control systems (VCS) are software tools that help software teams manage changes to source code over time by keeping a track of modifications done in the code. A VCS can be used to store code as well as configuration files, documentation, data files, or any other content that we may need to track.

One of the most popular VCS is GIT. Git is a VCS created by Linus Torvalds in 2005. The developer who launched the Linux kernel. Git is a free open source system available for installation on Unix, Windows and mac-OS based platforms. Linus originally created Git to help manage the task of developing the Linux kernel. This task was difficult because many geographically distributed programmers were collaborating to write a bunch of code. Linus had requirements for system operation and performance that were not met by the VCS tools of the day. So he decided to write his own.

Unlike some VCSs that are centralized around a single server, Git has a distributed architecture. This means that each person contributing to a repository has a complete copy of the repository on their own development machine. Contributors

can share and take back changes made by others as needed. And because the repositories are all local to the computer used to create the files, most operations can be done very quickly. If a developer wants to collaborate with other people, it's usually a good idea to set up a repository on a server that will act as a sort of hub that everyone can interact with. As mentioned, Git doesn't depend on any kind of centralized server to provide control organizations to its workflow. It can run as a standalone program as a server and as a client; that means it can be used on a single machine without having a network connection or as a server on a machine where to host a repository. And Git can be used as a client to access the repository from another machine or even the same one.

Git clients can communicate with Git servers on the network using HTTP, SSH, or Git's special protocol. Git can be used with or without a network connection. It can be used for small projects with a single developer or large projects with thousands of contributors. It can be used to track private work or share the work with others by hosting code on public servers like Github, Gitlab or others.

I chose Git as VCS in my project because of its popularity, cross-platform support, and robust feature set. However, there are many other tools that can be used to accomplish the same task. Other VCSs like Subversion or Mercurial are in the market [11].

### **3.3 Jenkins: a CI tool**

Jenkins is an open source CI tool written in Java. It is used by teams of all sizes, for projects in a wide variety of languages and technologies, including .NET, Ruby, Groovy, Grails, PHP and others, as well as Java.

Jenkins is one of the most popular tools for continuous integration. It helps in fetching the updated code and preparing an executable build. This tool comes with hundreds of open source plugins that are quick and easy to install. These plugins cover everything from version control systems, build tools, code quality metrics, build notification, integration with external systems, user interface customization, and more [12].

### 3.3.1 Jenkins Architecture

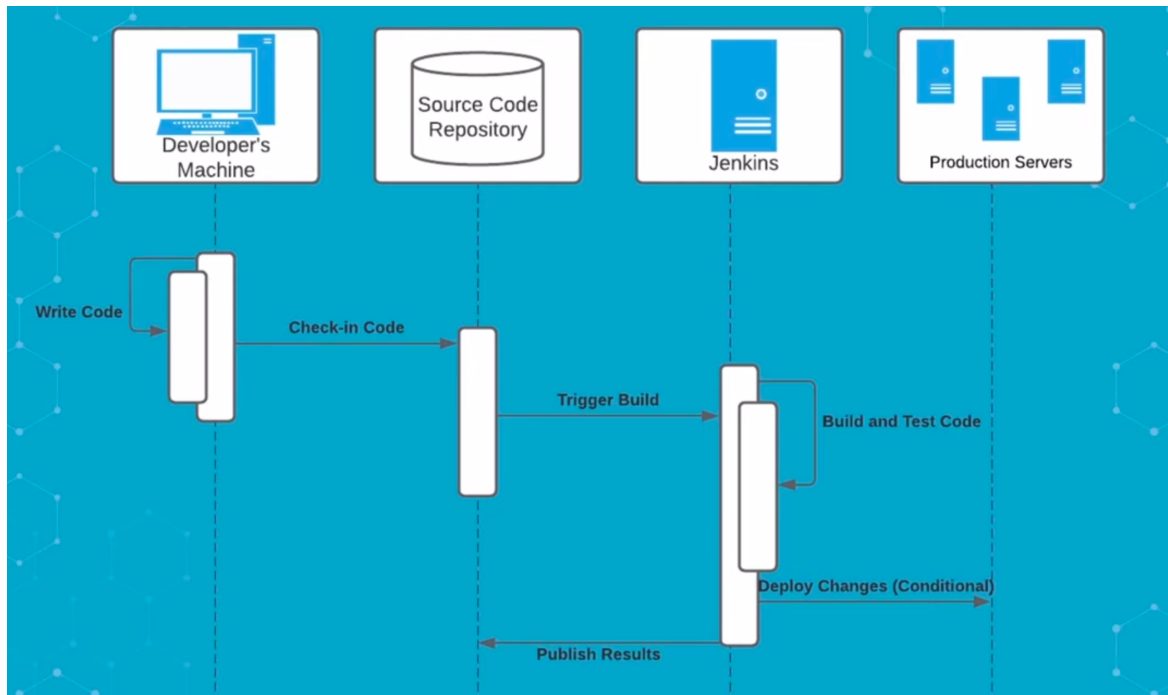


Figure 3.1: Jenkins Architecture [6]

The developer starts by writing code, when the code is ready to be reviewed and tested, he checks his code in a source code repository (GitHub, Gitlab, Bitbucket, etc...). When the developer registers his code in the repository, it triggers a build in Jenkins. The build output will be available in the Jenkins Dashboard. Automatic notification can also be sent to the developer [13].

In the diagram, the communication goes from the source code repository to Jenkins, but this is not always the case. Jenkins could be configured to pull the source code repository for new changes.

Whenever Jenkins detects new changes in a particular repository, it also triggers a build. Once a build is triggered in Jenkins, it will then pull the developer's code changes to its local workspace where it will build and test the code automatically using the latest version of the code base.

If the automated build and test steps are successful, Jenkins will automatically deploy those changes to the production environment. If the build and test steps fail, the automated deployment will not occur. Whether the build succeeds or fails, Jenkins will always publish the results to the source code repository.

### 3.3.2 Jenkins Installation

Jenkins will be installed on a server where the central build will take place. After downloading the Jenkins war file, I'm going to start the Jenkins server. Once it's up and running, one can access Jenkins from the link **http://localhost:8080**, this link will bring up the Jenkins Dashboard (see Appendix A).

### 3.3.3 Jenkins Plugins

Jenkins advantages is its extensibility with plugins. These plugins support the automation of various development tasks and were primarily developed to satisfy the need for CI and CD. The development tasks that these plugins support include-but are not limited to- running tests, static code analysis, building projects and deployment, among many, many other capabilities.

### 3.3.4 Jenkins Configuration

The most basic function of any CI tool is to monitor source code in a version control system and retrieve and build the latest version of your source code whenever changes are made. In this case, I'm going to use Git, therefore I need to install the Git Plugin. First, I'll go to the **System Configuration** screen (figure A-1) then I'll click on **Manage Plugins**

## 3.4 Set up the repository

After installing the Jenkins server, it's time to setup the repository that I'll be working on. Therefore, I'm going to clone a remote repository into my local machine so that I can collaborate on a project with a team. For that I'm using the "git clone" command.

## 3.5 A Jenkins Job Build

For this section, we'll create a job in Jenkins which fetches a simple HelloWorld application, builds and run the java program.

The following steps are illustrated with detailed captures in the appendix C

1. This Jenkins Job is a freestyle project (figure C.1). The freestyle project is the central feature of Jenkins Jenkins will build the project, combining with a VCS (figure C.2) .

2. We need to specify the location of the files which need to be built. In this example, we have a GitHub repository which contains a 'helloWorld.java' file so we'll copy its SSH URL alongside with the credential which is a private token that we already talked about in the configuration section (figure C.3). Then, we'll choose the build triggers (figure C.4).  
After that, we have to add a webhook in the Github repository, it's the URL of the Jenkins server with "/github-webhook/" right after.  
Now, we are going to set the Git build trigger and it's going to be "the push" event (figure C.5).
3. Now that I set up the repository am going to upload a the program to the remote repository then commit. This will instantly trigger a build in Jenkins. During the build, Jenkins generates a console output at runtime on which we can see the results of the build in the console log section as shown in the figure. below.

## Console Output

```
Started by GitHub push by ayacherigui
Running as SYSTEM
Building in workspace /var/lib/jenkins/workspace/Jenkins-Job
The recommended git tool is: NONE
using credential GitHubPrivateKey
Cloning the remote Git repository
Cloning repository git@github.com:ayacherigui/Jenkins-Job.git
> git init /var/lib/jenkins/workspace/Jenkins-Job # timeout=10
Fetching upstream changes from git@github.com:ayacherigui/Jenkins-Job.git
> git --version # timeout=10
> git --version # 'git version 2.25.1'
using GIT_SSH to set credentials
> git fetch --tags --force --progress -- git@github.com:ayacherigui/Jenkins-Job.git +refs/heads/*:refs/remotes/origin/* #
timeout=10
> git config remote.origin.url git@github.com:ayacherigui/Jenkins-Job.git # timeout=10
> git config --add remote.origin.fetch +refs/heads/*:refs/remotes/origin/* # timeout=10
Avoid second fetch
> git rev-parse refs/remotes/origin/main^{commit} # timeout=10
Checking out Revision 3478e3545e41c36e551db3cd3d041fb2a31fe60d (refs/remotes/origin/main)
> git config core.sparsecheckout # timeout=10
> git checkout -f 3478e3545e41c36e551db3cd3d041fb2a31fe60d # timeout=10
Commit message: "Delete AppTest.java"
First time build. Skipping changelog.
Finished: SUCCESS
```

Figure 3.2: Console output

## 3.6 Conclusion

This example shows the value of a CI/CD pipeline, even on a small scale. Certainly, we haven't explored half of Jenkins' functionality, but we were able to highlight its key features. This sets the stage for future work where I will implement a CI/CD pipeline on a larger scale using deployment automation with tools like Docker and Kubernetes and with more sophisticated testing.

# General Conclusion

CI/CD is a way to leverage automation in the agile product development process. This model is a key part of adopting a DevOps culture, which in turn is designed to guarantee faster time to market and customer satisfaction. To keep up with the speed and agility of DevOps, security should be involved throughout the development lifecycle, from planning to monitoring. To meet these demands, organization teams needs to automate some of the security processes and practices, and that's where DevSecOps comes in.

DevSecOps is the integration and automation of security to help DevOps teams to find problems earlier and mitigate them quickly while saving time and money. One of the most important security practices is the integrating of security testing into the CI/CD pipeline.

The CI/CD pipeline offers several benefits for software development. These include reducing the number of code changes, speeding up mean time to resolution, improving test reliability, accelerating the rate of release, reducing the number of software backlogs and increasing customer satisfaction. Unfortunately, attackers exploit weaknesses in the CI/CD pipeline and other DevOps infrastructure. They can steal information, mine crypto-currencies and inject malware into software.

DevSecOps helps eliminate the bottleneck caused by legacy security models and tools on the modern CI/CD pipeline. It also bridges the gap between IT and security while ensuring efficient and secure code production.



# Bibliography

- [1] Wikipedia, the free encyclopedia. software-development-life-cycle, 2019. [Online; accessed /2019/04/08/].
- [2] Wikipedia, the free encyclopedia. Waterfall vs agile, 2019. [Online; updated /2019/04/08/].
- [3] Wikipedia, the free encyclopedia. The devops cycle, 2020. [Online; updated /2021/3/11/].
- [4] J. Lord. *Building CI/CD Systems Using Tekton: Develop flexible and powerful CI/CD pipelines using Tekton Pipelines and Triggers*. Packt Publishing, 2021.
- [5] Continuous integration, 2021. [Online; accessed /2022/01/10/].
- [6] Jenkins architecture, 2021. [Online; updated /2020/08/23/].
- [7] G. Kim, J. Humble, P. Debois, and J. Willis. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. ITpro collection. IT Revolution Press, 2016.
- [8] J. Bond. *The Enterprise Cloud: Best Practices for Transforming Legacy IT*. O'Reilly Media, 2015.
- [9] J.F. Smart. *Jenkins: The Definitive Guide: Continuous Integration for the Masses*. O'Reilly Media, 2011.
- [10] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. A Martin Fowler Signature Book. Addison-Wesley, 2010.
- [11] J. Loeliger and M. McCullough. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. Oreilly and Associate Series. O'Reilly Media, Incorporated, 2012.

- [12] M. Labouardy. *Pipeline as Code: Continuous Delivery with Jenkins, Kubernetes, and Terraform*. Manning, 2021.
- [13] B. Laster. *Jenkins 2: Up and Running: Evolve Your Deployment Pipeline for Next Generation Automation*. O'Reilly Media, 2018.

# Appendix



# Appendix A

## Jenkins Dashboard



Figure A.1: Jenkins dashboard



# Appendix B

## Jenkins Configurations

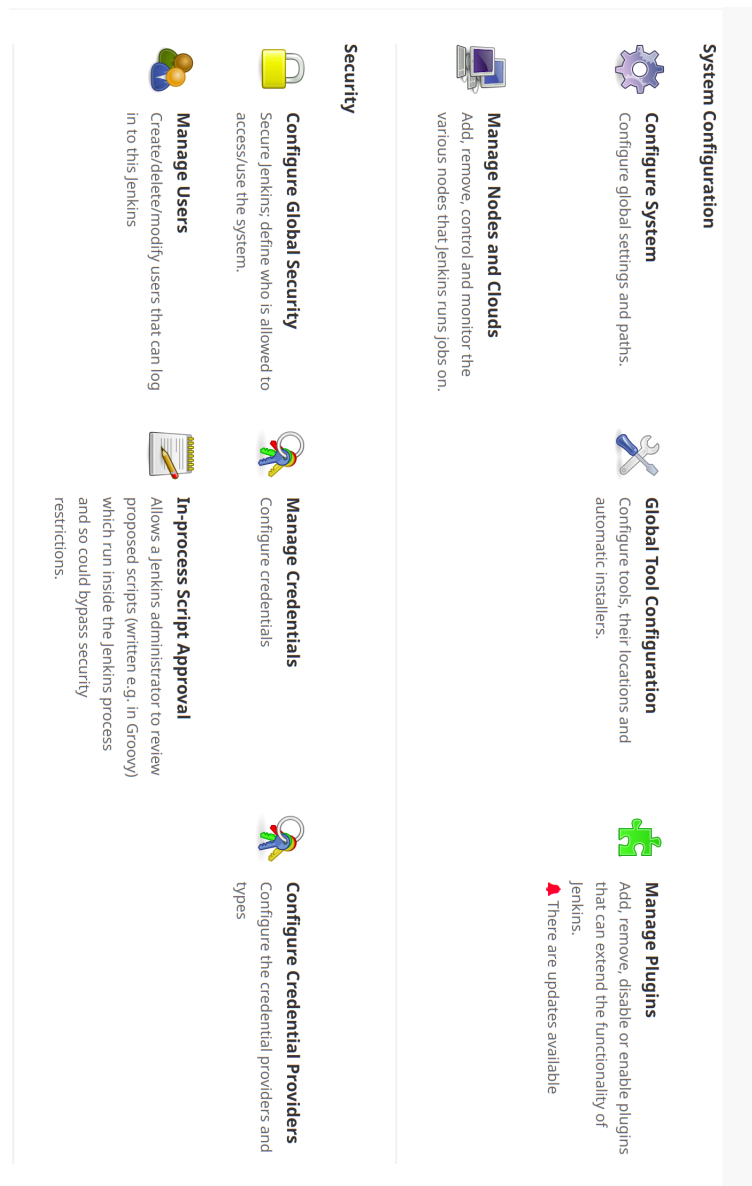


Figure B.1: System Configuration screen





# Appendix C

## Jenkins Job Build

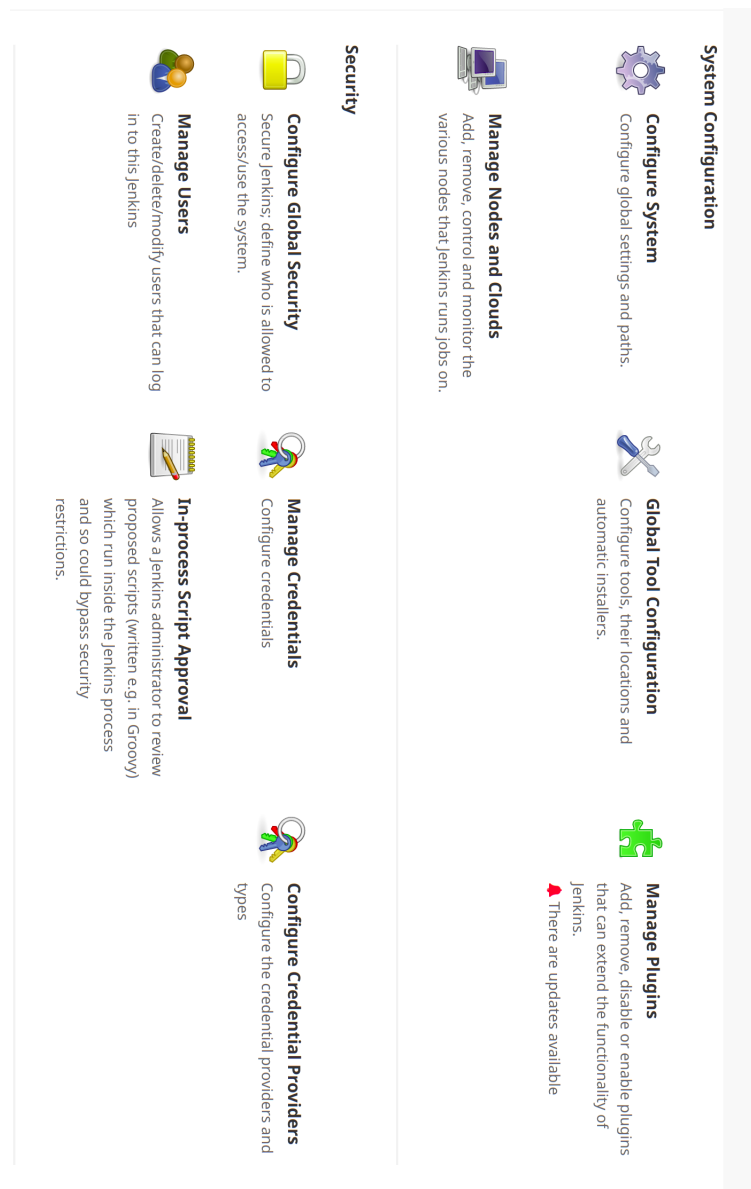



Figure C.1: System Configuration screen


### Enter an item name

Jenkins-Job


» Required field

**Freestyle project**


This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

**Maven project**


Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

**Pipeline**

Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

**Multi-configuration project**

Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

**Folder**

Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.

Figure C.2: Jenkins Project Choices

General

Source Code Management

Build Triggers

Build Environment

Build


Post-build Actions

### Source Code Management


☐ None  
☒ Git ?



**Repositories ?**

Repository URL ?

 Please enter Git repository.

**Credentials ?**

- none - 

 Add 

Advanced...

Figure C.3: Source Code Management integration

GeneralSource Code ManagementBuild TriggersBuild EnvironmentBuildPost-build Actions

Additional Behaviours

Add ▾

Build Triggers

☐ Trigger builds remotely (e.g., from scripts) ?

☐ Build after other projects are built ?

☐ Build periodically ?

☒ GitHub hook trigger for GITScm polling ?

☐ Poll SCM ?

Build Environment

☐ Delete workspace before build starts

☐ Use secret text(s) or file(s) ?

☐ Abort the build if it's stuck

☐ Add timestamps to the Console Output

☐ Inspect build log for published Gradle build scans

☐ With Ant ?

Build

Add build step ▾

Figure C.4: Build Triggers

**Webhooks / Add webhook**

We'll send a POST request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, x-www-form-urlencoded, etc). More information can be found in [our developer documentation](#).

---

**Payload URL \***

<http://8dd2-197-238-248-254.ngrok.io/>

**Content type**

application/json

**Secret**

---

**Which events would you like to trigger this webhook?**

☒ Just the push event.

☐ Send me **everything**.

☐ Let me select individual events.

Figure C.5: Git Webhook Settings