# Packaging and Release along with DevOps

# Table of Contents

# 1. Overview of DevOps

Unit 1.1 - Introduction to DevOps

Unit 1.2 - Basic of Virtualization

## Key Learning Outcomes

**At the end of this module, you will be able to:**

1. Explain the need for DevOps.
2. Explain the various SDLC models.
3. List and explain the various DevOps stages.

# UNIT 1.1: Introduction to DevOps

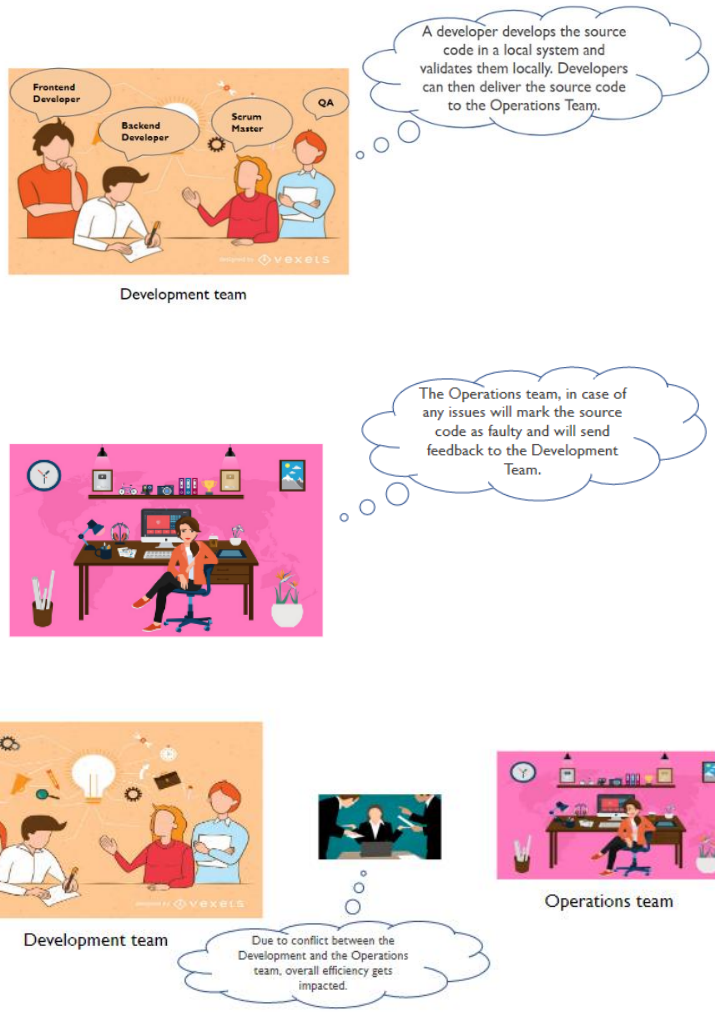
## Unit Objectives

**At the end of this unit, you will be able to:**

1. Explain what DevOps is and why we need it.
2. Explain what SDLC is.
3. List the benefits of SDLC.
4. Describe the various phases in SDLC.
5. Discuss various DevOps stages.

## 1.1.1 Evolution of DevOps

### 1.1.1.1 Software Development Before DevOps



## 1.1.1.2 Software Development After DevOps


## 1.1.2 Introduction to DevOps

DevOps is a software engineering method and practice that combines software development (Dev) and software operation (Ops). DevOps strives for (shorter development cycles, increased deployment frequency, and more reliable releases) in tandem with business goals. The main elements of the DevOps movement are the promotion of automation and the observation of all the steps in software construction. DevOps is an organizational capability concentrated on the agile relationship among the business development, test, and operations organizations.

## 1.1.2.1 Ingredients of DevOps

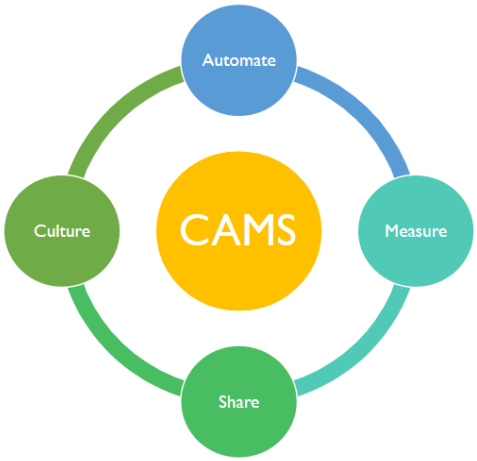



*Fig 1.1.2.1 Ingredients of DevOps*

**C stands for Culture:** Culture is the most important part of the DevOps movement. Accomplishing this takes a shift in culture, which in turn takes a shift in the organization and the process.

**A stands for Automation:** Automation isn't only about writing shell scripts; it is also about exempting the engineers from the mundane.

**M stands for Measurement**: Measurement is about monitoring and tracking performance, so there is feedback. Errors occur, but the goal is to learn once and for all.

**S stands for Sharing:** The major success factor of DevOps in any organization is sharing the tools, discoveries, and knowledge among the teams.

## 1.1.2.2 DevOps Principles

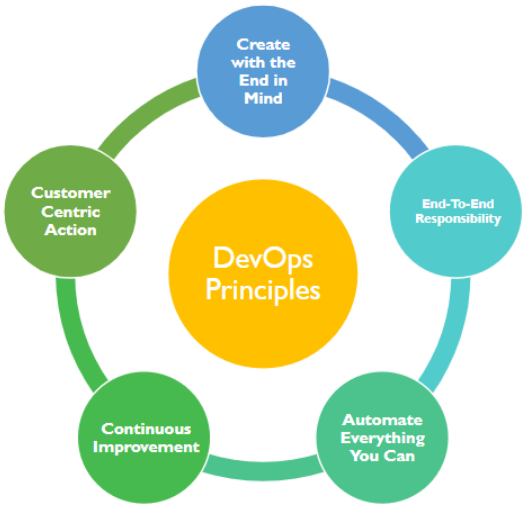



*Fig 1.1.2.2 DevOps Principles*

**Customer Centric Action:** It is critical to have short feedback loops with real customers and end users nowadays, which aids in the development of IT products and services centered on these clients.

**Create with the End in Mind:** Organizations need to let go of waterfall and process-oriented models, where each unit or individual works only for a particular role or function without overseeing the complete picture.

**End-To-End Responsibility:** In a DevOps environment, teams are vertically organized such that they are fully accountable for everything.

**Continuous Improvement:** Organizations need to adapt continuously in the light of changing circumstances (e.g., when customer needs change or new technology becomes available).

**Automate Everything You Can:** Think of automation as not only the software development process (continuous delivery, including continuous integration and continuous deployment), but also the process of automating the whole infrastructure landscape by implementing more and by using DevOps tools.

## 1.1.3 DevOps Adopted Sectors


*Fig 1.1.3 DevOps Adopted Sectors*

## 1.1.3.1 Benefits of DevOps

**Accelerated Time-to-Market:** With the DevOps method of approach, the time to develop a fully functional software product is reduced. Processes that used to take weeks and lots of handoffs are drastically reduced to a few button clicks or running a script instead.

**Cost Minimization:** The organizations that have already shifted to the DevOps method of approach report a 20% cost reduction on average.

**Customer Satisfaction:** With the DevOps method of approach, customer satisfaction will also be higher since they get on time delivery with uncompromised quality. This will almost certainly boost business, and the overall situation is favorable.

**Reliable Releases:** In a DevOps environment, the whole team is responsible for releasing new features and app improvements. The combination of a shared code base, continuous integration, and automated testing results in more reliable releases.

**Quick Bug Fixing:** Improved communication and cooperation. It coordinates between departments to organize bug fixes at any stage of development.

## 1.1.4 Challenges of DevOps

**Difficulties with Integration:** While performing integration, the team needs training for the staff members.

**Automated Testing:** Lots of organizations prefer to implement DevOps, but it cannot happen without automated testing. Automated testing is one of the essential parts of the approach.

**Relatively High Costs:** The integration and exploitation of the DevOps approach involve huge investments of both time and money.

**Tools Compatibility:** If your product is not compatible with DevOps tools, it will be very difficult to implement DevOps.

## 1.1.5 DevOps Tool Chain

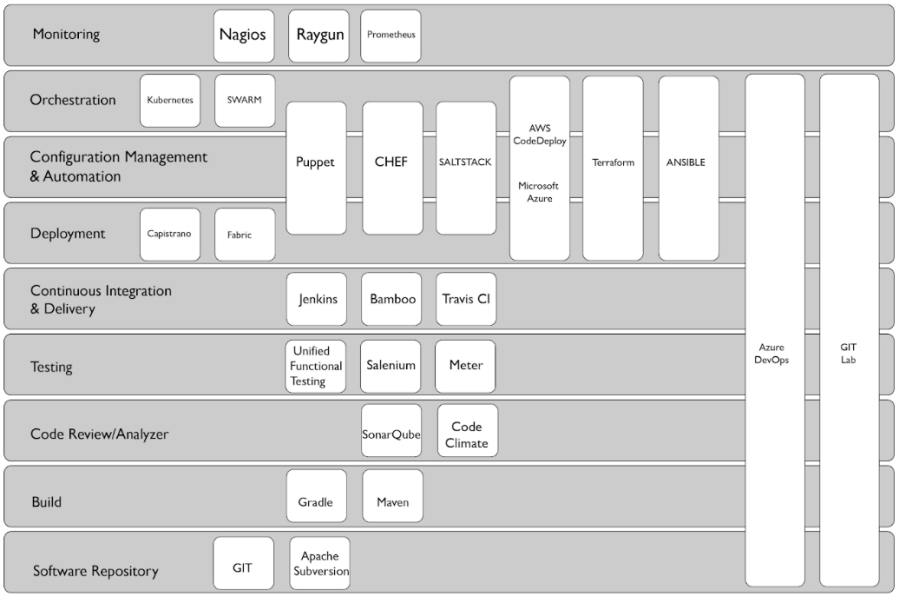



*Fig 1.1.5 DevOps Tool Chain*

## 1.1.6 Agile Methodology

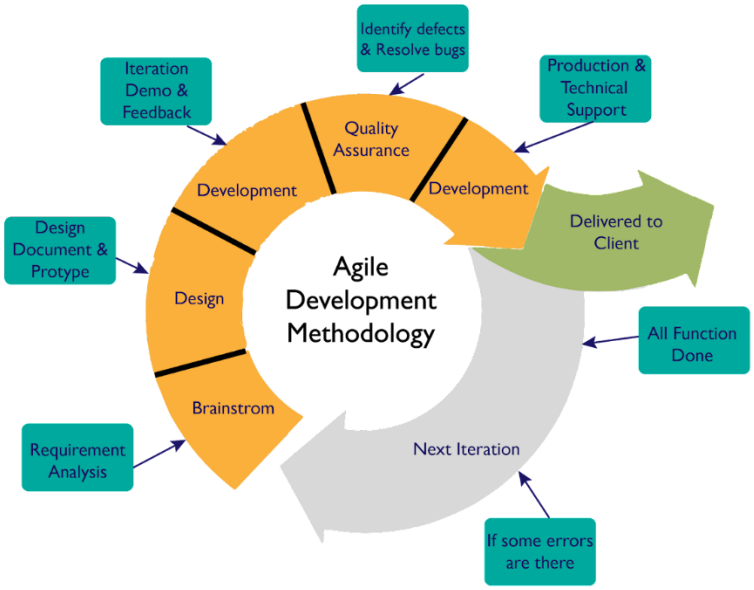



*Fig 1.1.6 Agile Methodology*

Around 2001, a group of experienced software developers like Kent Beck, Martin Fowler, Ron Jeffries, Ken Schwaber, and Jeff Sutherland came up with the Agile Manifesto. This manifesto was documented with their shared beliefs on what a modern software development process should look like. They endorsed collaboration over documentation, self-management over management practices, and single decision making. It was also intended to make the whole development process faster and to make more frequent releases to clients. Agile software development methodology is one of the simplest and most iterative approaches to managing projects and has the capability to convert a business need into software solutions. The goal of agile development is to enable faster deliveries with high efficiency and quality. It also focuses on how business requirements can be easily met. The ultimate value of Agile development is that it enables teams to deliver value faster with greater quality and predictability and greater aptitude to respond to change. Scrum and Kanban are two of the most widely used Agile methodologies.

## 1.1.6.1 Scrum

Scrum is one of the agile development methods that focuses on empowering development teams and advocates working in smaller teams (of 7-9 members). Scrum is mostly used to manage complex software development using iterative and incremental practices. Scrum significantly increases efficiency and makes release cycles shorter as compared to the waterfall model. It consists of three roles:

**Scrum Master:** It is responsible for setting up the team and the sprint meetings and managing obstacles.

**Product Owner:** It creates product backlogs and prioritizes them.

**Scrum Team:** It manages overall sprints and sprint cycles.

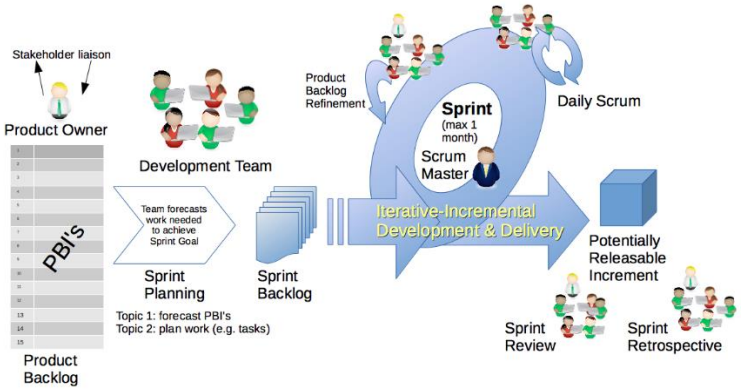



*Fig 1.1.6.1 Scrum*

## 1.1.6.2 Kanban

Kanban was developed by Toyota engineers to make manufacturing efficient. It's a popular framework used to implement agile and DevOps software development. It consists of a Kanban board, where all work items are represented, and it allows the team members to see the status of each piece of work. It aims to help you visualize your work, maximize efficiency, and improve continuously.

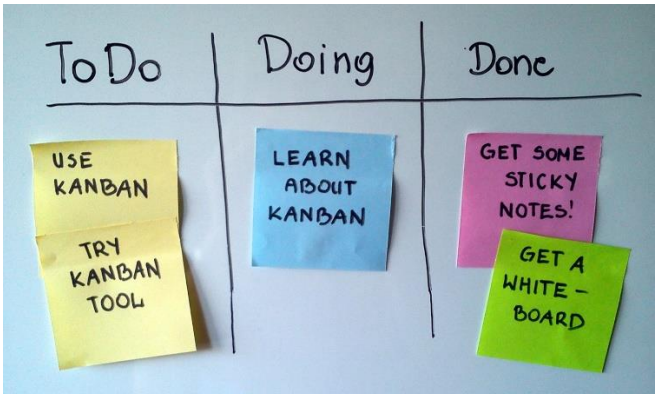



*Fig 1.1.6.2 Kanban*

## 1.1.6.3 Scrum Vs Kanban

| | Scrum | Kanban |
|---|---|---|
| Roles and Responsibilities | Each team member has a predefined role, Scrum Master, Product Owner, and Scrum Team | No specific teams |
| Delivery Timelines | Deliveries are planning as sprints and within that sprint all work should be completed | Products are delivered continuously and on demand. |
| Changing Requirements | During Sprint changing requirements are strongly discouraged | Allows changing requirements prior to completion of project |
| Best Implementation | Best for teams whose priorities may not change over time | Best for projects with widely varying priorities |
| Metrics | Velocity | Cycle time throughput |

## 1.1.6.4 Agile Values

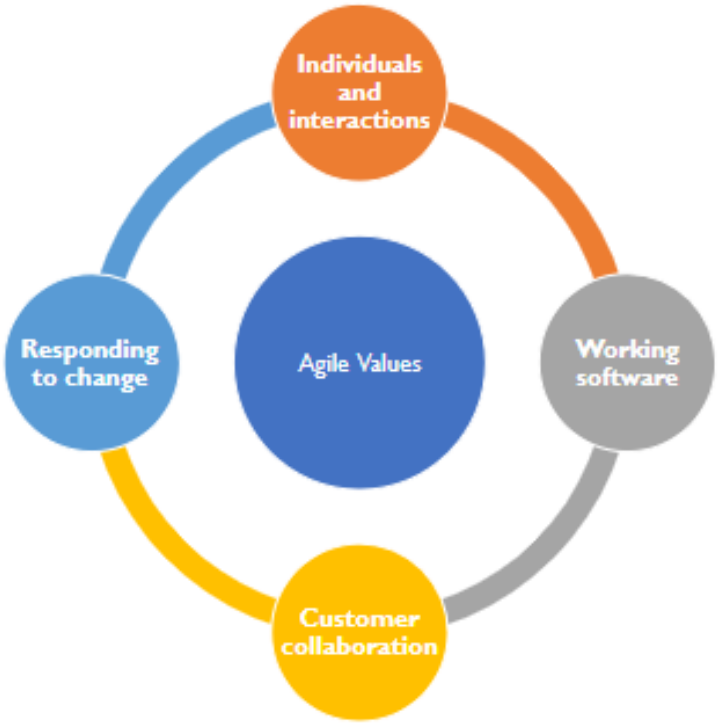



*Fig 1.1.6.4 Agile Values*

## 1.1.6.5 Agile Principles

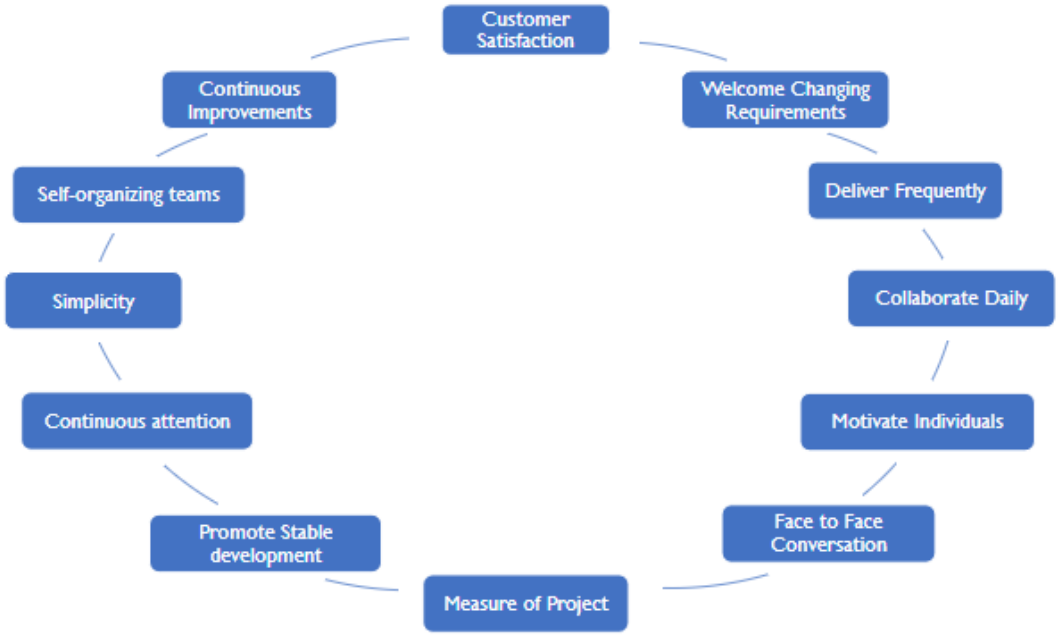



*Fig 1.1.6.5 Agile Principles*

## 1.1.7 Software Development Models

There are 4 types of software development model

- Waterfall model
- V-model
- Iterative and incremental model (Agile)
- Spiral model

## 1.1.7.1 Waterfall Model

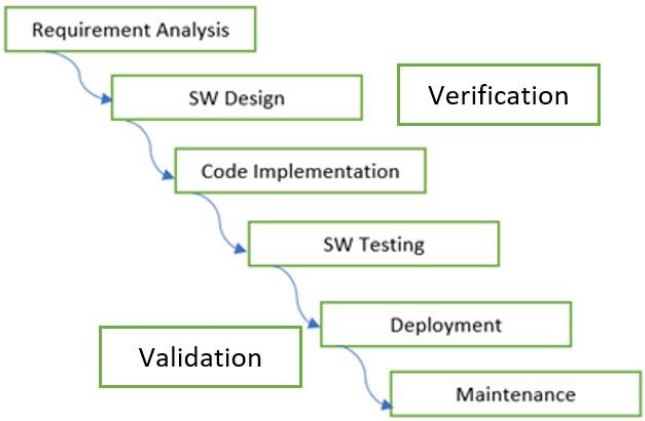



*Fig 1.1.7.1 Waterfall Model*

**Pros**:

- Step-by-step verification and validation

**Cons**:

- Ambiguous
- Static
- High risk
- Misinterpretation of requirements can cost project resources and budget.

## 1.1.7.2 V-Model

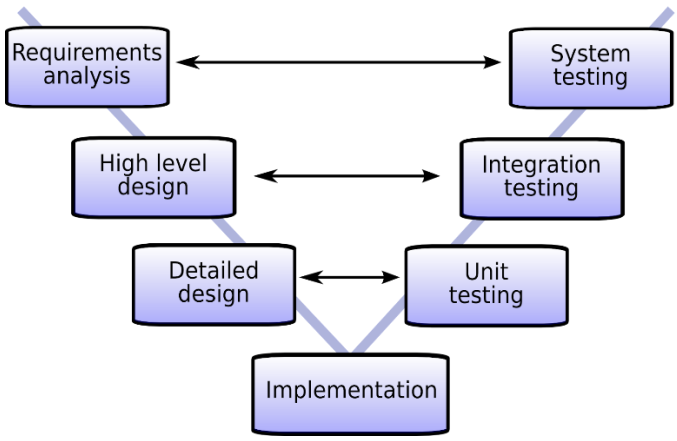



*Fig 1.1.7.2 V-model*

**Pros**:

- Artifacts related to validation are prepared during the verification phase (SYR ↔ SYT and SWR ↔ SWT).
- Avoids the flow of defects from one stage to another.
- More suitable for firmware application development.

**Cons**:

- No consideration was given to changing requirements
- Rigidity in the framework and not flexibility.
- Hard to use for prototype developments.

## 1.1.7.3 Agile Model

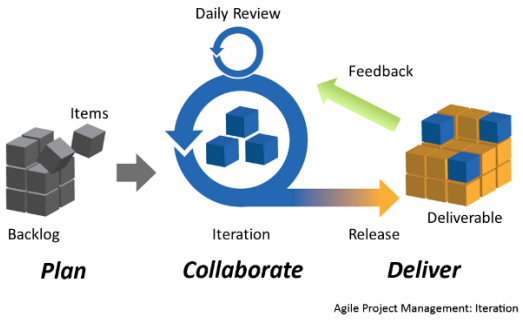



*Fig 1.1.7.3 Agile Model*

**Pros**:

- Incremental and iterative approach
- Working software for every iteration
- Highly responsive to changing requirements

**Cons**:

- Workflow coordination is difficult
- Dedicated availability of resources is the key challenge to overcome
- Applicable for smaller team sizes

## 1.1.7.4 Spiral Model

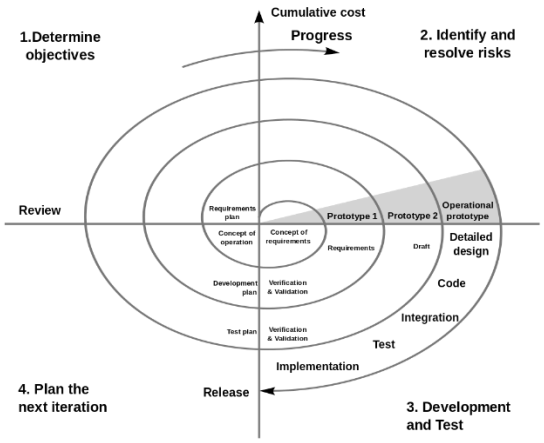



*Fig 1.1.7.4 Spiral Model*

**Pros**:

- Requirements can be captured more accurately.
- Users see the system early.
- Development can be divided into different portions, and the risky portion can be developed earlier for better risk management.

**Cons:**

- It is not appropriate or cost-effective for small, low-risk projects.
- The process is complex.
- The spiral may go on indefinitely.

## 1.1.7.5 Waterfall Vs Agile

| Waterfall | Agile |
|---|---|
| Software development is divided into sequential phases | Agile follows an iterative approach to development |
| It is quite rigid | It is flexible and can be managed to any extend |
| This model involves large teams | It promotes small and mid-sized teams |
| It requires customers only at certain phase | It allows customers to be available throughout project |
| There is no prioritization of features | Features are prioritized which increase efficiency of product |
| It responds slower to changing requirements | It responds quickly to changing requirements |
| With waterfall model feedback takes more time to reach to development team | It helps with faster feedback to development team |

## 1.1.7.6 SDLC Vs Agile/DevOps

| | Agile | DevOps |
|---|---|---|
| Team Composition | Development, QA, and Product Owner | Development and Operations |
| Goals | Bring Development, QA, and Product Owner Team Closer | Bring Development and Operations team closer to increase efficiency |
| Duration | 2-4 weeks sprints | Continuous process |
| Feedback | Provided by customers and Product Owners | Internal feedback between Development and Operations teams |
| Tools | Git, JIRA, Lean Kit, | Jenkins, Docker, Kubernetes, Ansible, Puppet, ELK |

## 1.1.8 Software Development Life Cycle

### 1.1.8.1 Developer's Mindset

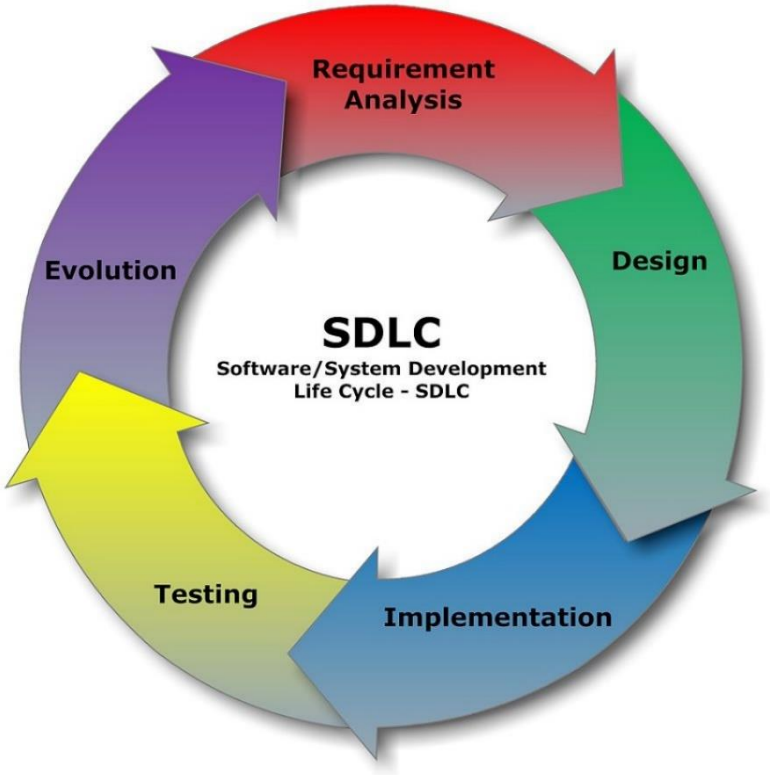



*Fig 1.1.8.1 Software Development Life Cycle*

SDLC, or Software Development Life Cycle is the sequence of various activities and processes followed for the development of a SW product. The activities include

**Requirement Analysis:** Perform requirement analysis for system-level requirements and derive SWR.

**Design:** Understand the architectural runway for the proposed requirements and work with SWA to check the feasibility of the architecture to be implemented for code development.

**Implementation:** Implement the SW Code for the SWRs, following coding standards and guidelines for specific industries.

**Testing:** Analyse the test reports generated from various validation phases and fix the bugs for different SW releases.

**Evolution:** Repeat the process for various SW releases by keeping track of proposed milestones.

### 1.1.8.2 Collaboration

The idea behind DevOps is that development, operations, and other functions must work closely together by cooperating and collaborating. Breaking down organizational silos improves communication among the teams, which gives everyone access to information about what was done in the past, the people involved in a particular work, and the associated results. It helps with decision making, which in turn produces better output and better ideas. It also helps teams improve their skills by collaborating with each other.

## 1.1.9 DevOps Stages

### 1.1.9.1 DevOps Implementation

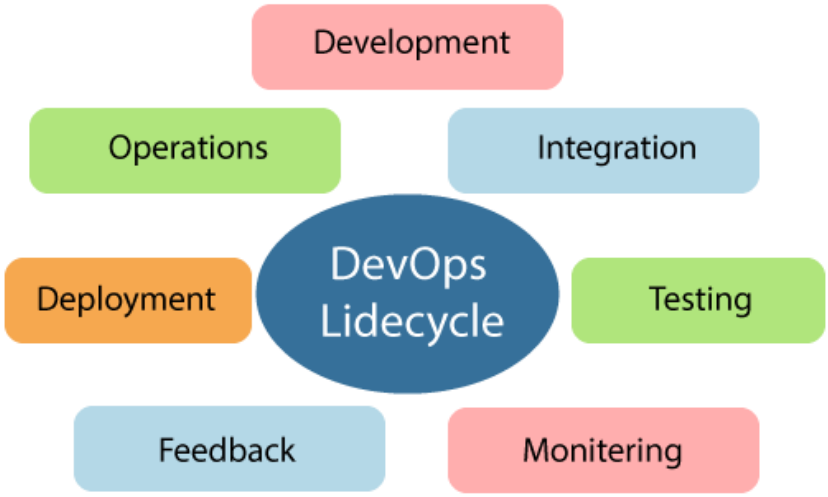



*Fig 1.1.9.1 DevOps Implementation*

**Continuous Development:** Software planning and coding are involved in this phase. The vision of the project is decided during the planning phase, and the developers begin developing the code for the application.

**Continuous Integration:** This is the core of the whole DevOps process. In this software development process, the developers need to make changes often. That may be on a daily or weekly basis. If any early problems are detected, they can be detected during the build. It also includes unit testing, integration testing, code review, and packaging.

**Continuous Testing:** In this phase, the developed software is continuously tested for bugs. Automation testing tools such as TestNG, JUnit, Selenium, etc., are used for constant testing. These tools allow QAs to test multiple code bases thoroughly in parallel to ensure that there are no flaws in the functionality. This entire testing phase can be automated with the help of a continuous integration tool called Jenkins.

**Continuous Monitoring:** Monitoring is a phase that involves all the operational factors of the entire DevOps process, where important information about the use of the software is recorded and carefully processed to find out trends and identify problem areas.

**Continuous Feedback:** By analysing the results of the software's operations, the application development is constantly improved. This is carried out by placing the critical phase of constant feedback between the operations and the development of the next version of the current software application.

**Continuous Deployment:** In this phase, the code is deployed to the production servers. Also, it is essential to ensure that the code is correctly used on all the servers. The new code is deployed continuously, and configuration management tools play an essential role in executing tasks frequently and quickly.

**Continuous Operations:** All DevOps operations are based on continuity with complete automation of the release process and allow the organization to continuously accelerate the overall time to market.

## 1.1.10 Automation in DevOps

Automation enables faster execution throughout the Software Development Life Cycle (SDLC), keeping up with the speed of DevOps. Automation can be employed and extended to code development, middleware configuration, database and networking changes, and essential testing, including regression testing and load testing. Automation saves the time and effort of the developers, testers, and operations personnel, and, in turn, reduces total costs. Automation will also help to remove any manual errors from the overall process, which will increase both efficiency and quality of delivery.

# UNIT 1.2: Basic of Virtualization

## Unit Objectives

**At the end of this unit, you will be able to:**

1. Explain the basic Linux commands.
2. Illustrate the various Linux file systems.
3. Differentiate the bare metal, virtual machine, and container.

## 1.2.1 What is Linux?

Linux is a well-known open-source operating system. The term "GNU Linux" refers not only to the Linux kernel but also to the set of programs, tools, and services that are typically bundled together with the Linux kernel to provide all the necessary components of a fully functional operating system. Linux-based operating systems are Arduino, Ubuntu, Fedora, Debian, and Mandriva.

**Why Linux?**

- Multi-user/Multitasking/Time-sharing concept.
- Portability
- Modularity
- File structure
- Security
- Strong networking support
- Advanced graphics.

## 1.2.2 Linux Basic Command

- **PWD (Print Working Directory):** This command is to find out the pathname of the current directory.
    - Syntax
        - % PWD
- **ls (List):** This command is to find out what is in the home directory.
    - Syntax
        - % ls
    - Example
        - $ ls –m To list files across the page, separated by commas.
        - $ ls ~ List the contents of your home directory by adding a tilde after the ls command.
        - $ ls / List the contents of your root directory.

- **mkdir (Make directory):** This command is to create a subdirectory in the current directory.
  - Syntax
    - % mkdir “foldername”
  - Example
    - $ mkdir new
- **cd (Change directory):** This command is to change the current working directory to another.
  - Syntax
    - % cd “new folder name”
  - Example
    - $ cd newdir (changing current folder to a new folder named newdir.)
    - $ cd.. (This command will take you to the folder up in the directory.)
- **cat (Concatenate)**: This command is used to display the contents of a file on the screen.
  - Syntax
    - % cat filename.txt
  - Example
    - $ cat one.txt
- **head:** This command is used to display the first ten lines of a file on the screen.
  - Syntax
    - % head filename.txt
  - Example
    - $ head one.txt
- **tail:** This command is used to display the last ten lines of a file on the screen.
  - Syntax
    - % tail filename.txt
  - Example
    - $ tail one.txt
- **mv (Move):** This command is used to move the content from one file to another and also to move file to another directory.
  - Syntax
    - % mv oldfilename newfilename
    - % mv filename directoryname
  - Example
    - $ mv file1.txt file2.txt
    - $ mv file.txt /home/new/test
- **cp (copy):** This command is used to copy the content from one file to another file.
  - Syntax
    - % cp file1name file2name
  - Example
    - $ cp file1.txt file2.txt

- **echo:** This command is used to print text on the standard output.
  - Syntax
    - % echo message
  - Example
    - $ echo hello world
- **sudo:** This command allows the user to execute a command as the superuser or another user.
  - Syntax
    - % sudo command
  - Example
    - $ echo hello world
- **clear:** This command is used to clear the screen.
  - Syntax
    - % clear
  - Example
    - $ clear

## 1.2.3 Linux File System

**What are file systems?**

A file system is the way the files are organized or stored on the disk. In other words, the file system is the method and data structure that an operating system uses to keep track of files on a disk or partition. There are many kinds of file systems; each one has a different structure and logic and has properties such as speed, flexibility, security, size, and more. Linux supports numerous file systems, but the common choices for the system disk on a block device include the ext* family (ext2, ext3, and ext4), XFS, OpenZFS, and BtrFS.

## 1.2.3.1 File Systems

**Ext:**

This is an old file system introduced in the Linux operating system, also called an extended file system. It provides a basic file system for Linux, using virtual directories to handle physical devices and to store data on the physical device in fixed-length blocks. It uses a system called inodes to track information about the files stored in the virtual directory. The inode creates a separate table on each physical device called the inode table to store the file information. Linux refers to each inode in the inode table using a unique number, assigned by the file system as data files are created.

**Ext2:**

The original ext file system had quite a few limitations, such as limiting files to only 2 GB in size. The upgraded version of the ext file system is called ext2. It is an expansion of the basic abilities of the ext, but with the same structure. The ext2 file system expands the inode table format to track additional information about each file on the system. The ext2 inode table adds the created, modified, and last accessed time values for files to help the system administrator track file access. This file system helps reduce fragmentation by allocating disk blocks in groups when you save a file.

**Ext3:**

This file system was added to the Linux kernel in 2001. It uses the same inode table structure as the ext2 filesystem but adds a journal file to each storage device to journal the data written to the storage device. By default, the ext3 file uses the ordered mode method of journaling, only writing the inode information to the journal file but not removing it until the data block has been successfully written to the storage device. The ext3 file system doesn't provide any recovery from accidental deletion of files, and there's no built-in data compression available. The ext3 file system doesn't support encrypting files.

**Ext4:**

This file system was introduced in the Linux kernel in 2008 and is now the default file system used in most popular Linux distributions. To support compression and encryption, the ext4 file system also supports a feature called extents. Extents allocate space on a storage device in blocks and only store the initial block location in the inode table. This file system also incorporates block pre-allocation.

**XFS:**

It is a 64-bit file system that was first introduced in 1994 and built into the Linux kernel in 2001. It is the default file system for RedHat Linux. XFS supports a maximum file size of 8 EiB and restricts filename length to 255 bytes. It supports journaling, and like ext4, it saves the changes in a journal before they are committed to the main file system. This reduces the possibility of file corruption. Data is structured in B+ trees, providing efficient space allocation and increased performance.

**OpenZFS:**

ZFS is a file system that provides a way to store and manage large volumes of data, but you must manually install it. ZFS on Linux does more than file organization, so its terminology differs from standard disk-related vocabulary. It is free to install ZFS on Linux, and it provides robust storage with features such as on-the-fly error correction; disk-level, enterprise-strength encryption; transactional writes -- writing all or none of the data to ensure integrity; use of solid-state disks to cache data; and use of high-performance software rather than proprietary RAID hardware.

**BtrFS:**

Oracle designed BtrFS, which stands for B-tree File System, and released it in 2009 with the Linux kernel. BtrFS supports a maximum file size of 16 EiB and limits the maximum filename length to 255 characters. Some of the BtrFS' features are online defragmentation, online block device addition and removal, RAID support, compression configurable per file or volume, file cloning, checksums, and the ability to handle swap files and swap partitions.

## 1.2.4 Bare Metal Vs Virtual Machine Vs Container

**Bare Metal**

The traditional way of running an application was directly on dedicated hardware, i.e., one machine, one operating system (OS), and one service or task. It can provide the best performance since there isn't a virtualization layer between the host and the libraries and apps. The physical resources were highly underutilized, could not be partitioned easily for different teams or apps, and were difficult to scale.

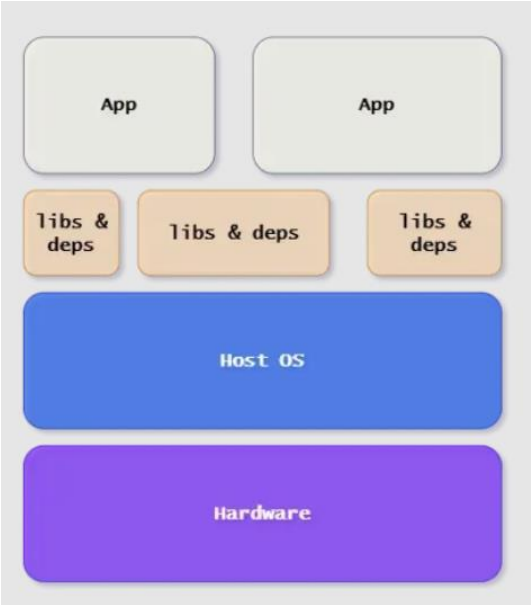



*Fig 1.2.4 Bare Metal*

**Virtual Machine**

A software implementation of a machine that executes programs like a physical machine. A virtual machine provides an interface identical to the underlying bare hardware. The operating system creates the illusion of multiple processes, each executing on its own processor with its own memory.

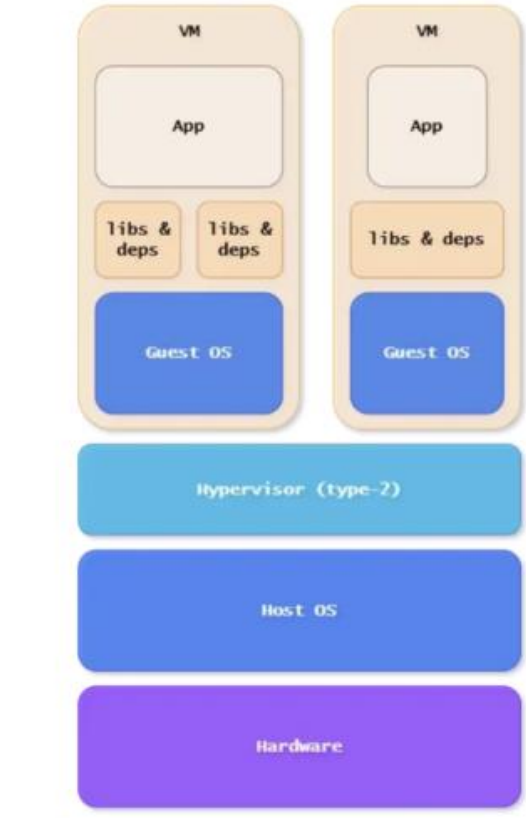



*Fig 1.2.4 Virtual Machine*

**Containers**

A container is an environment that runs an application that is not dependent on the operating system. It isolates the app from the host by virtualizing it. This allows users to create multiple workloads on a single OS instance. They don't need a guest OS or hypervisor; instead, all containers running on a host machine share the OS kernel of the host system and only contain the application(s) and their libraries and dependencies. Additionally, since containers are not concerned with the underlying hardware, they can be run on a myriad of different platforms.

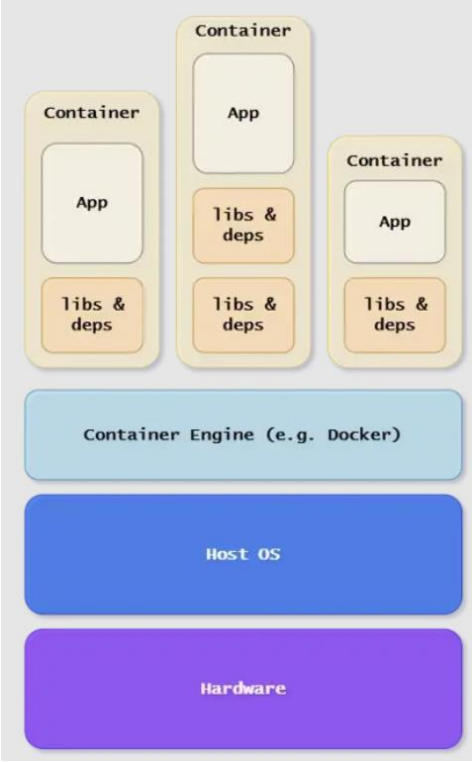



*Fig 1.2.4 Containers*

# 2. Version Control with Git

Unit 2.1 - Version Control System

Unit 2.2 - Git and GitHub

## Key Learning Outcomes

**At the end of this module, you will be able to:**

1. Explain the GNU make file and its setup
2. Explain the Make file creation.
3. Explain GIT and GitHub
4. Illustrate various GIT methods
5. Publish the code in Git Repo

# UNIT 2.1: Version Control System

## Unit Objectives

**At the end of this unit, you will be able to:**

1. Explain the GNU make file setup and configuration
2. Explain the makefile creation.

## 2.1.1 GNU Make and Makefiles

### 2.1.1.1 GNU Make

GNU Make is a tool that controls the generation of executables and other non-source files of a program from the program's source files. Make gets its knowledge of how to build your program from a file called the makefile, which lists each of the non-source files and how to compute them from other files.

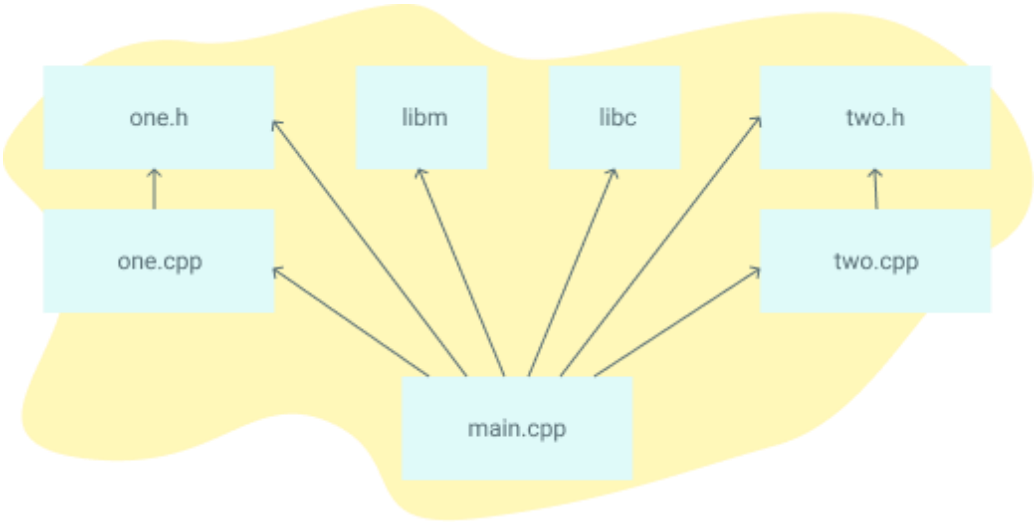



*Fig 2.1.1.1 GNU Make*

### 2.1.1.2 Makefile Syntax

Makefile consists of a set of rules that looks like this

```
1 targets: prerequisites
2     command
3     command
4     command
```

*Fig 2.1.1.2 Makefile rules*

The targets are file names, separated by spaces. Typically, there is only one per rule. The commands are a series of steps typically used to make the target (or targets). These need to start with a tab character, not a space. The prerequisites are also listed by their full names, separated by spaces. These files need to exist before the commands for the target are run. These are also called dependencies.

## 2.1.1.2 Makefile Example

Let's start with a hello world example:

```
hello:
    echo "Hello, World"
    echo "This is make command example"
```

We have one target called hello. This target has two commands. This target has no prerequisites. We'll then run make hello. The commands will run if the hello file does not exist. If hello exists, no commands will run. It's important to realize that "hello" can be both a target and a file. It's because these two are interconnected. When a target is run, the command will create a file with the same name as the target. In the previous example, the hello target did not create the hello file.

Let's create a more typical Makefile – one that compiles a single C file. Before that, we need to create a file called hello.c.

```
//hello.c

int main()
{
    return 0;
}
```

Create a Makefile called Makefile (as always).

```
hello:
    gcc hello.c -o hello
```

This time, try simply running make. Since there's no target supplied as an argument to the make command, the first target is run. In this case, there's only one target (hello). The first time you run this, blah will be created. The second time, you'll see make: 'hello' is up to date. That's because the hello file already exists. But there's a problem: if we modify hello.c and then run make, nothing gets recompiled.

We can solve that by adding a prerequisite

```
hello: hello.c
    gcc hello.c -o hello
```

When we run make again, the first target is selected because the first target is the default target. This has a prerequisite of hello.c. Make decisions if it should run the hello target. It will only run if hello doesn't exist or hello.c is newer than hello.

The following Makefile eventually executes all three targets. When you run make in the terminal, it will build a program called hello in a series of steps:

- Make selects the target hello because the first target is the default target
- hello requires hello.o, so make searches for hello.o target
- hello.o requires hello.c, so make searches for the hello.c target
- hello.c has no dependencies, so the echo command is run
- The cc -c command is then run, because of all of the hello.o dependencies are finished
- The top cc command is run because all the blah dependencies are finished
- That's it: blah is a compiled c program.

```
hello: hello.o
    gcc hello.o -o hello //Runs third

hello.o: hello.c
    gcc -c hello.c -o hello.o //Runs second

//Typically blah.c would already exist, but I want to limit any additional required files

hello.c:
    echo "int main() { return 0; }" > hello.c //Runs first
```

All three targets will be rerun if hello.c is deleted. If you edit it (and thus change the timestamp to something newer than hello.o), the first two targets will run. If you run touch hello.o (and thus change the timestamp to newer than blah), then only the first target will run. If you change nothing, none of the targets will run.

## 2.1.1.3 Make Clean

Clean is often used as a target that removes the output of other targets, but it is not a special word in Make. Clean is doing two new things here:

- It's a target that is not first (the default) or a prerequisite. That means it'll never run unless you explicitly call to make it clean.
- It's not intended to be a filename. If you happen to have a file named clean, this target won't run, which is not what we want.

```
some_file:
    touch some_file

clean:
    rm -f some_file
```

### 2.1.1.4 Make Variables

Variables can only be strings. You'll typically want to use: =, but = also works.

```
files := file1 file2
some_file: $(files)
    echo "Look at this variable: " $(files)
    touch some_file

file1:
    touch file1
file2:
    touch file2

clean:
    rm -f file1 file2 some_file
```

Single or double quotes have no meaning to Make. They are simply characters that are assigned to the variable. Quotes are useful to shell and bash, though, and you need them in commands like printf. In this example, the two commands behave the same:

```
a := one two // a is assigned to the string "one two"
b := 'one two' // Not recommended. b is assigned to the string "'one two'"
all:
    printf '$a'
    printf $b
```

## 2.1.2 Targets

**The all target:** If you want to make multiple targets and you want all of them to run, then make an all target. Since this is the first rule listed, it will run by default if make is called without specifying a target.

```
all: one two three

one:
    touch one
two:
    touch two
three:
    touch three

clean:
    rm -f one two three
```

**Multiple targets:** When there are multiple targets for a rule, the commands will be run for each target. $@ is an automatic variable that contains the target name.

```
all: f1.o f2.o

f1.o f2.o:
    echo $@

// Equivalent to:
// f1.o:
//    echo f1.o
// f2.o:
//    echo f2.o
```

## 2.1.3 Commands and Execution

**Command Echoing/Silencing:** Add an @ before a command to stop it from being printed. You can also run make with -s to add an @ before each line.

```
all: one.c two.c three.c
    gcc one.c two.c three.c -o
    @gcc one.c two.c three.c -o make -I.
```

**Command Execution:** Add an @ before a command to stop it from being printed. You can also run make with -s to add an @ before each line.

```
all:
    cd ..
    // The cd above does not affect this line, because each command is effectively run in a new shell
    echo `pwd`

    // This cd command affects the next because they are on the same line
    cd ..;echo `pwd`

    // Same as above
    cd ..; \
    echo `pwd`
```

## 2.1.4 Double Dollar Sign

If you want a string to have a dollar sign, you can use $$. This is how to use a shell variable in bash or sh. In the following example, take note of the differences between Makefile variables and Shell variables.

```
make_var = I am a make variable
all:
    // Same as running "sh_var='I am a shell variable'; echo $sh_var" in the shell
    sh_var='I am a shell variable'; echo $$sh_var

    // Same as running "echo I am a amke variable" in the shell
    echo $(make_var)
```

## 2.1.5 Error Handling with –k, -I, and -

Add –k when running to make sure to continue running even in the face of errors. Helpful if you want to see all the errors in Make at once. To suppress the error, place a - before the command. To make this happen for every command, add -i.

```
one:
    // This error will be printed but ignored, and make will continue to run
    -false
    touch one
```

## 2.1.6 Recursive Use of Make

To recursively call a makefile, use the special $(MAKE) instead of make. It will pass the make flags for you and won't be affected by them.

```
new_contents = "hello:\n\ttouch inside_file"
all:
    mkdir -p subdir
    printf $(new_contents)
    cd subdir && $(MAKE)

clean:
    rm -rf subdir
```

## 2.1.7 Conditional Part of Makefiles

**Conditional if/else**

```
foo = ok

all:
ifeq ($(foo), ok)
    echo "foo equals ok"
else
    echo "nope"
endif
```

**Check if a variable is empty**

```
nullstring =
foo = $(nullstring) // end of line; there is a space here

all:
ifeq ($(strip $(foo)),)
    echo "foo is empty after being stripped"
endif
ifeq ($(nullstring),)
    echo "nullstring doesn't even have spaces"
endif
```

**Check if a variable is defined**

```
bar =
foo = $(bar)

all:
ifdef foo
    echo "foo is defined"
endif
ifndef bar
    echo "but bar is not"
endif
```

## 2.1.8 Functions

**First Functions:**

Functions are mainly just for text processing. Call functions with $(fn, arguments) or ${fn, arguments}. You can make your own using the call built-in function. Make has a decent amount of built-in functions.

```
bar := ${subst not, totally, "I am learning make file"}
all:
    @echo $(bar)
```

If you want to replace spaces or commas, use variables

```
comma := ,
empty:=
space := $(empty) $(empty)
foo := a b c
bar := $(subst $(space),$(comma),$(foo))

all:
    @echo $(bar)
```

Do not include spaces in the arguments after the first. That will be seen as part of the string.

```
comma := ,
empty:=
space := $(empty) $(empty)
foo := a b c
bar := $(subst $(space), $(comma) , $(foo))

all:
    // Output is ", a , b , c". Notice the spaces introduced
    @echo $(bar)
```

**if function:**

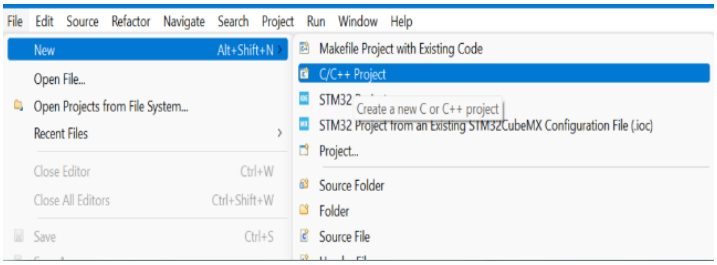
if checks if the first argument is non-empty. If so, runs the second argument, otherwise run the third.

```
foo := $(if this-is-not-empty,then!,else!)
empty :=
bar := $(if $(empty),then!,else!)

all:
    @echo $(foo)
    @echo $(bar)
```

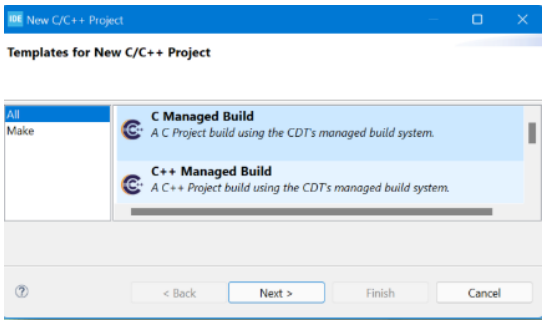
## 2.1.9 Makefile in STMCubeMX IDE

### 2.1.9.1 Creating Makefile in STMCubeMX IDE

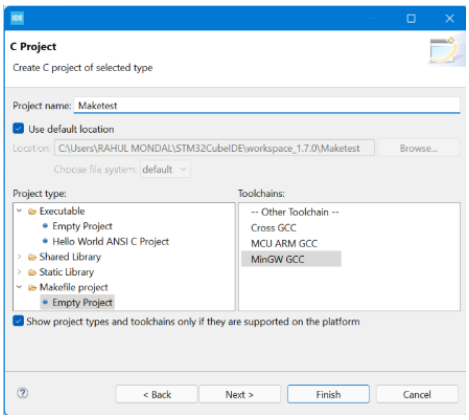
**Step 1:**




**Step 2:**




**Step 3:**



**Create the build files**

**main.c**

```c
#include<stdio.h>
#include "func.h"

int main()
{
    func();
return 0;
}
```

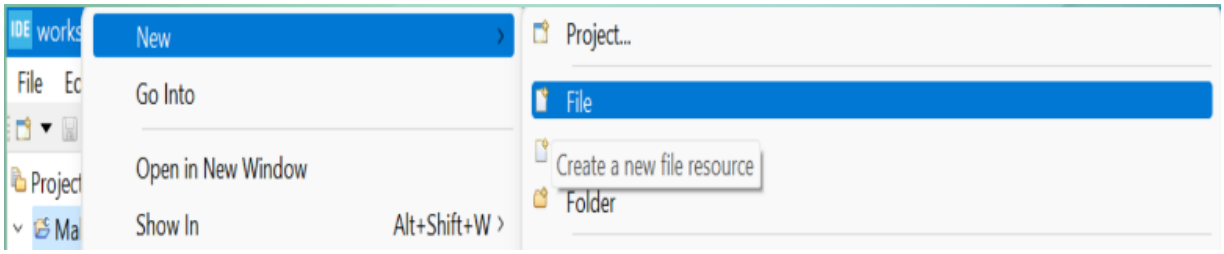
**func.c**

```c
#include "func.h"

void func()
{
    printf("Make file Example");
}
```

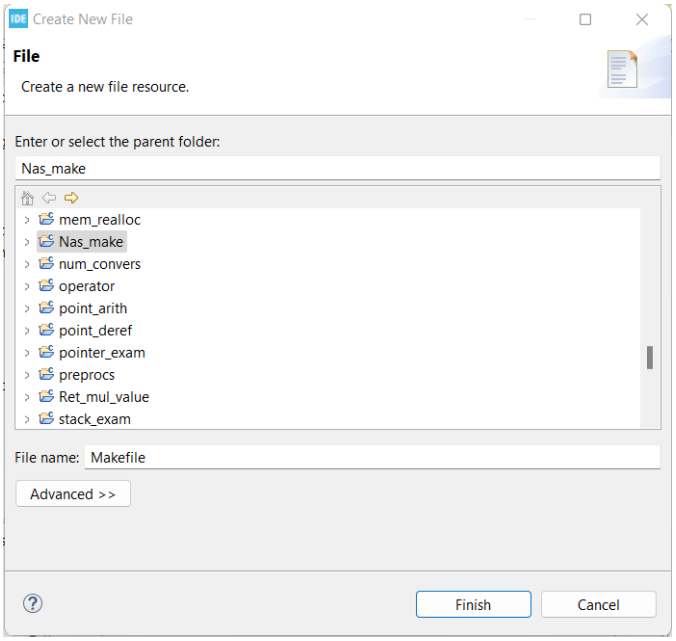
**func.h**

```c
#ifndef FUNC_H_
#define FUNC_H_

void func();

#endif /* FUNC_H_ */
```

**Create new file**



**The custom make file name should be “Makefile” only.**




**Custom makefile command**

```
1 all: main.c welcome.c func.c
2     gcc main.c welcome.c func.c -o make -I.
```

# UNIT 2.2: Git and GitHub

## Unit Objectives

**At the end of this unit, you will be able to:**

1. Explain what Git and GitHub are.
2. Illustrate the various Git methods.
3. Build and deploy the code in GIT Repo.

## 2.2.1 Introduction to GIT

Git is an open-source distributed version control system used to track modifications done to the source code during software development. It can be used to make changes even when the computer is turned off. It is much more efficient and faster when compared with a centralised version control system. It also supports branching, which can be used by different developers to work on multiple features simultaneously. It is one of the most secure forms of version control system that supports the cryptographic method Secure Hash Algorithm 1 (SHA-1).

### 2.2.1.1 Git Installation

Git is one of the open-source version control systems. It's mainly available for three types of platforms, as listed below:

**Unix/ Linux:** Git is available for most Unix-based distributions like Debian, Ubuntu, CentOS, Red Hat, Solaris, etc.

**Windows:** Git is also available for the Windows platform, and we can easily install it on a Windows desktop or server.

**Mac OS:** Apple's Mac OS also supports the installation of Git. We can download a DMG file and install it.

### 2.2.1.2 Git Terminologies

**Master**: Master is the default branch in the case of Git.

**Clone**: Clone is used to clone a remote repository as a local repository on a developer's machine.

**Commit**: The commit command saves commits to the Git local repository.

**Push and Pull**: These two commands are used to push and pull changes from and to remote repositories, respectively.

**HEAD**: HEAD refers to the latest commit done in the Git repository.

## 2.2.2 Who has Adopted Git?

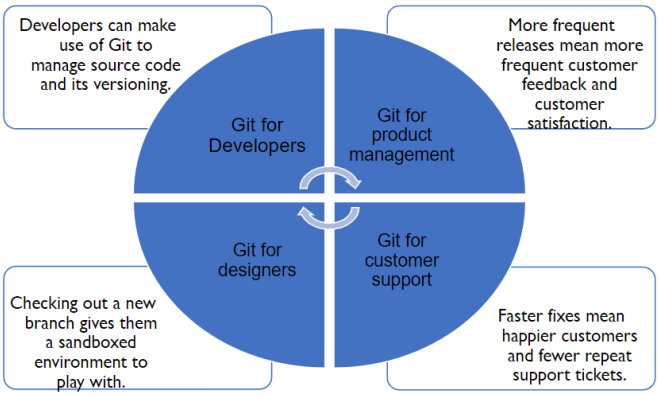



*Fig 2.2.2 Usage of Git*

## 2.2.3 Git Installation

```
root@ip-172-31-25-208:~# apt install git
Reading package lists... Done
Building dependency tree
Reading state information... Done
git is already the newest version (1:2.17.1-1ubuntu0.4).
0 upgraded, 0 newly installed, 0 to remove and 152 not upgraded.
root@ip-172-31-25-208:~# git --version
git version 2.17.1
root@ip-172-31-25-208:~#
```

*Fig 2.2.3 Git Installation*

Git is supported by most of the Unix based platforms. Some of the most common platforms are:

- Debian/Ubuntu
  - $ add-apt-repository ppa:git-core/ppa
  - $ apt update; apt install git
- Fedora/Red hat/CentOS
  - $ yum install git

Windows supports Git installation for both 32-bit and 64-bit architecture systems: Download the Git installer from https://git-scm.com/download/win and then proceed with the installation.


*Fig 2.2.3 Git Installation*

Mac is Apple's operating system, so we can follow two ways of doing Git installation on Mac.
DMG File https://git-scm.com/download/mac
Brew Command: Brew update and Brew install git


*Fig 2.2.3 Git Installation*

## 2.2.4 Git Configuration

**--global:** Global level configuration is user-specific; it applies to the operating system of the user. Global configuration values are stored in a file that is in the user's home directory.

**--system:** System-level configuration is registered across an entire machine. This covers all users in an operating system and all repos. The system-level configuration file is in a gitconfig file, off the system root path /etc/gitconfig on Unix systems.

To project Git config values in a global or local level project, the git config command is used as a convenience function. The git config command can accept arguments to state which configuration level to operate on.

**--local:** By default, git config will write to the local level if no configuration option is passed. Local configuration values are saved in a file called.git/config, which can be found in the repo's.git directory.

## 2.2.5 Git Workflow

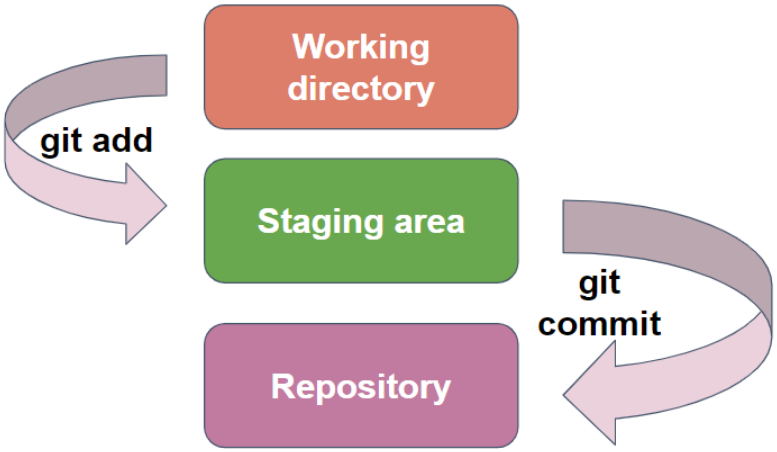



*Fig 2.2.5 Git Workflow*

The "Working Directory" is the directory where you are currently working. The "Staging Area" is when git starts tracking and saving modifications that occur in files. A local git repository is where all the commits will be saved. After modifications to the working directory, we can run the git add command to add files to the staging area (git add, git status). A local git repository is where all the commits will be saved. If we want to move changes permanently to a git repository, we must run the git commit command to save the changes (git commit -m "First Commit").

## 2.2.6 Git Init

To work with comparisons, we must perform some file setup. We must add some content to existing git repository files so that we can check the git diff between different indexes.

```
mkdir skill-lync
cd skill-lync
git init
git add. && git commit –m "Adding new content"
```

## 2.2.7 Git Add and Git Commit

After modifying "Working Directory," we can run the git add command to add files to the staging area. Let's start with dummy commits and then use git log to get the respective commit ids.

```
echo "First Commit" >> README.md
git add. && git commit –m "First Commit"
git log –n 2 --oneline     (To get last two commits ID)
git diff <COMMIT_ID1> <COMMIT_ID2>
```

## 2.2.8 Git Tracked Files

Tracked files are those files that are recorded under version control. We can add files to the version control system using the git add command. We can run the git status command to check the status of the current repo.

```
echo "New Content" >> README.md
git status
git add
git status
```

## 2.2.9 Recursive Add

We can use the git add command to add untracked files to the Git version control system. We can run the git add command on individual files or on a complete folder as well.

```
echo “Sample File" >> sample.txt
echo “Example File" >> example.txt
git status
Git add sample.txt          (Adding single sample.txt file to staging area)
git add --all               (Will add both Sample and example text file)
git status
```

## 2.2.10 Backing Out Changes

It's very simple to undo changes with the Git version control system if something goes wrong. The git reset command is used to revert changes made in the repository.

```
git status
git reset –hard
git status
```

## 2.2.11 Git Fetch

The git fetch command downloads commits, files, and refs from a remote repository into your local repo.Git fetch will only fetch content, but the local repository will not be merged with the new content; it remains in the same state as before.

```
git fetch <remote>
git fetch <remote> <branch>
git fetch –all
```

## 2.2.12 Git Stashing

Git Stash temporarily stashes changes done by the developer in the working copy. The git stash command takes your uncommitted changes, saves them away for later use, and then returns them from your working copy.

```
git status
echo "Adding Content" >> merge.txt
git status
git stash
git status
```

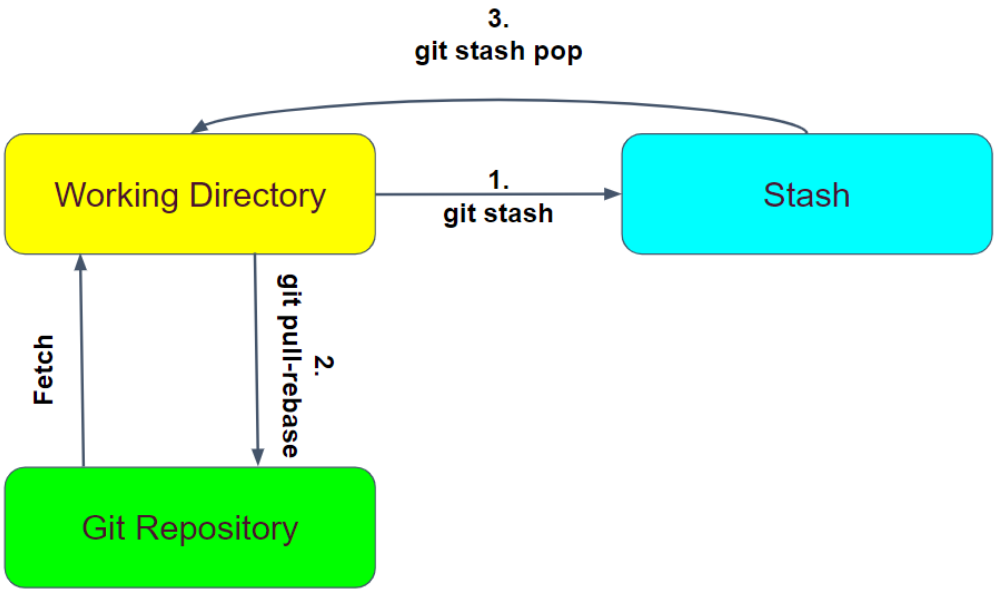



*Fig 2.2.12 Git Stashing*

## 2.2.13 Managing Stashes

With git, you can run stash multiple times, and all these stashes will be saved in your git repository. Any time you want to apply any of the stash, we can easily do it with git apply. By default, stashes are identified as "WIPs" (work in progress), besides the branch and commit that you created the stash from. We can add custom comments while saving a stash.

```
git stash save "stash done on master branch"
git stash list
git stash pop stash@{2}     (This will apply stash on the working index but remove stash.)
git stash apply stash@{2}   (This will apply to stash, but stash will still be in the repository.)
```

## 2.2.14 Git Cheat Sheet

**Git Config**: Used to set Git configuration values on global, system or local level.
**Git Init**: To initialized empty local repository.
**Git Add**: Used to add one or multiple files to staging area.
**Git Commit**: Permanently records changes stored in staging area to local repository.
**Git Reset**: Used to reset or back out changes done in local repository.
**Git Fetch**: Fetches all new commits, branches, and refs information from remote to local repository.
**Git Stash**: Stash will remove changes from local working area but will be preserved for later purpose.

## 2.2.15 Everything About Git

### 2.2.15.1 Git Renaming and Moving Files

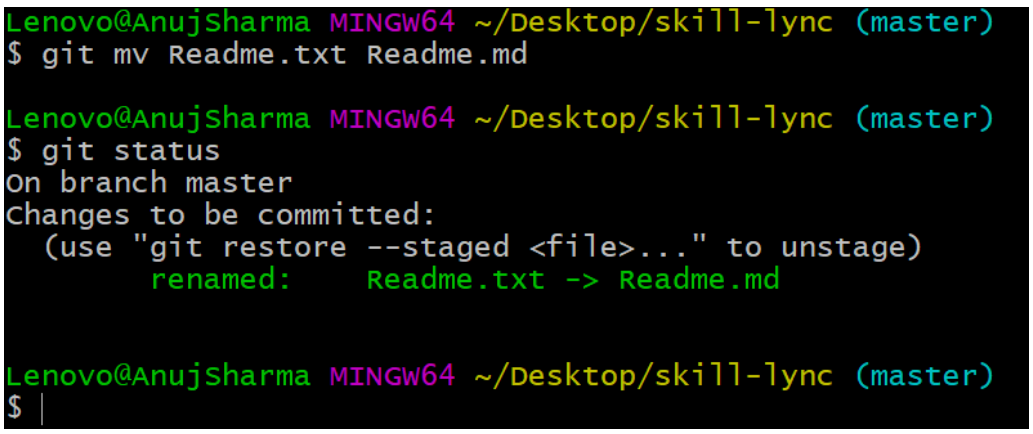



*Fig 2.2.15.1 Git Renaming and Moving Files*

We can use the git mv command to rename a file or move an existing file to another path in the repository.

```
git mv README.txt README.md
git status
```

## 2.2.15.2 Git Deleting Files

```
Lenovo@AnujSharma MINGW64 ~/Desktop/skill-lync (master)
$ git rm Readme.txt
rm 'Readme.txt'

Lenovo@AnujSharma MINGW64 ~/Desktop/skill-lync (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    Readme.txt


Lenovo@AnujSharma MINGW64 ~/Desktop/skill-lync (master)
$
```

*Fig 2.2.15.2 Git Deleting Files*

In the Git version control system, removing files is not a simple process of deleting them from the repository. We must use the git rm command to remove files from the repository.

git rm README.md

git status

## 2.2.15.3 Git History

```
Lenovo@AnujSharma MINGW64 ~/Desktop/skill-lync (master)
$ git log
commit 95e73ba11612e9f23ea3a7eaebcb08bc4b6cc4c2 (HEAD -> master)
Author: Anuj Sharma <abc@abc.com>
Date:   Wed Mar 17 13:39:22 2021 +0530

    Sample Content

Lenovo@AnujSharma MINGW64 ~/Desktop/skill-lync (master)
$ git log --oneline
95e73ba (HEAD -> master) Sample Content

Lenovo@AnujSharma MINGW64 ~/Desktop/skill-lync (master)
$
```

*Fig 2.2.15.3 Git History*

Git allows developers to track changes to your source code.

This gives you the power to check the history of all modifications performed to the repository.

git log

git log --oneline

## 2.2.15.4 Git Alias

```
Lenovo@AnujSharma MINGW64 ~/Desktop/skill-lync (master)
$ git config --global alias.last 'log -1 HEAD'

Lenovo@AnujSharma MINGW64 ~/Desktop/skill-lync (master)
$ git last
commit 95e73ba11612e9f23ea3a7eaebcb08bc4b6cc4c2 (HEAD -> master)
Author: Anuj Sharma <abc@abc.com>
Date:   Wed Mar 17 13:39:22 2021 +0530

    Sample Content

Lenovo@AnujSharma MINGW64 ~/Desktop/skill-lync (master)
$
```

*Fig 2.2.15.4 Git Alias*

An alias is considered a shortcut that allows us to use a simple command instead of long commands that are difficult to remember. These aliases can be used across multiple repositories, which helps the developer perform various git operations easily.

git config --global alias.last 'log -1 HEAD’

git last

## 2.2.15.5 Ignoring Files in Git

Git provides us with the feature to ignore some untracked files in the Git repository. We can mention file paths and file patterns that we want Git to ignore. A "gitignore" file is created as per the below steps to ignore or untrack files.

```
echo "File to be ignored" >> sample.txt
git status
echo "sample.txt" >> .gitignore
git status
```

## 2.2.16 Git Branching

Branching allows developers to deviate from the mainline without breaking the code in the mainline. A good branching strategy helps the developers collaborate easily without breaking the source code. Changes made in one branch do not impact the other branch. Developers can merge the changes from one branch to another once the code is tested and verified. Feature branches allow developers to work on multiple features at the same time, each isolated from the others. If a feature is not released, we don't have to rollback the changes. We can simply discard that specific feature branch. These branches help the developers work on their features in parallel. If we are not proceeding with the modifications, we can simply ignore or delete the branch from the git repository.

## 2.2.17 Git Merge

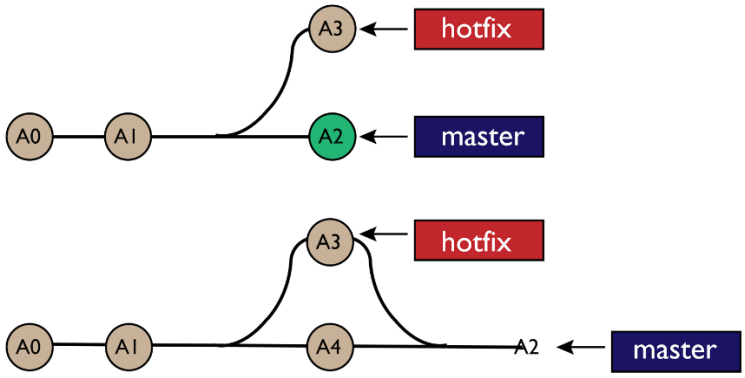



*Fig 2.2.17 Git Merge*

## 2.2.17.1 Git Merge Tools

While working with Git, we must frequently perform merging between multiple branches. To perform it efficiently and to resolve merge conflicts, we can use some GUI tools that can make life easy for the developers. There are multiple tools available that can help check diff and perform a visual merge of the source code.

- WinMerge
- GitKraken
- KDiff3
- Beyond Compare

## 2.2.17.2 Fast Forward Merges

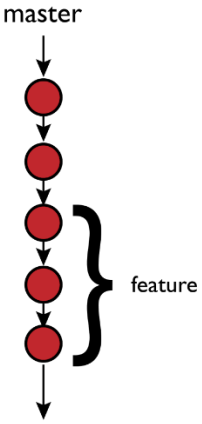



*Fig 2.2.17.2 Fast Forward Merges*

Git merge helps integrate changes from multiple branches into one. Git by default performs merging in fast-forward mode. The fast-forward method helps to keep git history linear without any distortions. With this method, no new commits get created in the branch logs, and we can easily delete these branches after merging.

**Fast Forward Merges: Commands**

git branch develop
git checkout develop
echo “Develop Branch Content" >> README.md
git add . && git commit -m “Develop Branch Commit“
git checkout master
git merge develop
git log --oneline -n 2 (This command will help you to see that there are no modifications done to the Git history)

## 2.2.17.3 Git Merge Without Fast Forward

Git merge without the fast-forward method is most suitable for staging or production branches. With this approach, instead of multiple commits, only one commit will be visible, which makes it quite easy to revert in case of a rollback. Also, from the git log history, we can easily identify the git merge commit.

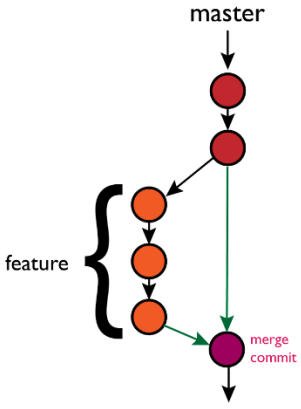



*Fig 2.2.17.3 Git Merge without Fast Forward*

**Git Merge without Fast Forward: Commands**

```
git branch develop
git checkout develop
echo “Develop Branch Content" >> README.md
git add . && git commit -m “Develop Branch Commit“
git checkout master
git merge develop ---no-ff
git log --oneline -n 2      (This command will help to see that there will be one new commit created in the Git history)
```

## 2.2.17.4 Git Pull Request

A git pull request is a mechanism for replicating the Git merging activity using a GUI rather than a command line. We can use the pull request feature, with which we can perform automatic merging without running any commands manually. We can even use this option to first compare changes between branches, and then a merge request can also be raised. Once the pull request is created and approved, we can even delete the source branch.

## 2.2.17.5 Git Rebase

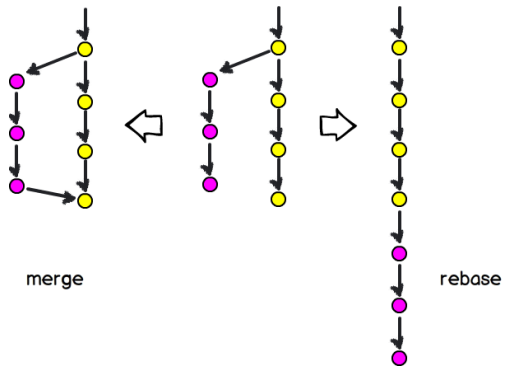



*Fig 2.2.17.5 Git Rebase*

Rebasing, just like merging, helps to sync up the changes made between two branches. The primary goal of rebasing is to make the Git repository history linear. Rebasing is the process of combining a sequence of commits into a new base commit. Rebase is used over merge in case the developers need a neat and clean Git history, since with rebase, history looks like local history only.

**Git Rebase: Example commands**

```
git init
echo "Adding content to master branch" > merge.txt
git add . && git commit -m "Adding content to master branch“
git checkout -b develop
echo "Adding content to Develop Branch" >> README.md
git add . && git commit -m "Adding to Develop branch“
git checkout master
git rebase develop      (Once this command is executed, you will see changes in the master branch)
```

## 2.2.17.6 Git Merge Conflicts

Merge conflicts occur when competing changes are made to the same line of a file or when multiple developers work on the same file at the same time. To solve a merge conflict caused by competing line changes, you should choose which changes to include from different branches in a new commit. Delete the conflict markers <<<<<<<, =======, >>>>>>> and make the changes you want in the final merge. In the case of merging, we can choose two different strategies while performing Git merging.

- git merge --strategy-option theirs or git merge --strategy-option ours

## 2.2.18 Git Tagging

Tags are references to a specific point in the Git history. Tagging is generally used to take a snapshot of the history and store it in the Git repository as a backup. A tag is like a branch, but unlike a branch, we cannot perform any changes to a tag. We usually create tags in case we want to release code from a specific branch.

- git tag <tag_name>

## 2.2.18.1 Annotated Tags

Annotated tags store extra metadata like author details, tag message, and date as full objects in the Git repository database. Like commits and commit messages, annotated tags have a tagging message. These tags were created to provide extra information related to tag messages, like commits and author details. We can run the below command, which will create a tag with extra details.

- git tag -a v1.0 -m "Git Tag Version 1.0"

## 2.2.18.2 Tagging a Git Commit

Git also allows us to generate a tag based on a specific commit. This enables the developers to create a tag that has a source code equivalent to the code in a specific commit. Use the below command to create a tag with a custom commit.

- git tag -a <tag_name> <commit_hash> -m "Message here"

## 2.2.18.3 Updating Tag with New Commit

When we create a tag from a branch, some commits get stored in the tag, which we cannot modify. In case we want to update the tag with a new commit or change the commits stored in a tag, we must use the "–f" flag to do so. Use the below Git tag command to update a tag with new commits.

- git tag -f <tag_name> <commit_hash>

## 2.2.19 Git Push

The git push command syncs the changes from the local repository to the remote repository. While performing push activity, developers need network connectivity. Git push helps the developers collaborate with each other by sharing the source code of the repository among themselves.

git push <remote>, git push <remote> <branch>

git push --all

## 2.2.20 Git Pull

The git pull command downloads the commits, files, and refs from a remote repository and merges them into your local repo. The git pull command internally executes Git fetch and Git merge to sync the changes from the remote repository to the local repository.

```
git pull <remote>
git pull <remote> <branch>
git pull –all
```

## 2.2.21 GitHub Vs Bitbucket Vs GitLab

| GitHub | Bitbucket | GitLab |
|---|---|---|
| GitHub provides free public and private repositories | Bitbucket provides free public and private repositories | GitLab provides free public and private repositories |
| It is an enterprise-licensed tool. | It provides an Atlassian-licensed tool. | It supports the community version, which is open source. |
| It is written in Erlang and Ruby. | It is written in Python and the Django web framework. | It is written in the Ruby programming language. |
| It has around 6,90,00,000 projects hosted. | It projects are mostly private and not available publicly. | It does not have many open source projects hosted. |
| It has around 56 million users associated with it. | It has around 50,00,000 users. | It has around 30 million registered users. |
| You can create free repositories of any size, even larger than 1 GB, on GitHub. | We cannot create repositories larger than 1 GB in Bitbucket. | It supports the maximum size of the repository, up to 5 GB. |

## 2.2.22 Git Remotes

Git repositories are remote locations where developers can collaborate and share source code with each other. GIT remote assists developers in configuring connections to remote repositories and establishing connectivity between local and remote repositories.

```
git remote -v
git remote add origin <repo_url>
git remote remove origin
```

## 2.2.23 Git Cheat Sheet

**Git mv**: Used to rename a file or move a file.
**Git rm**: Used to remove a file from a repository.
**Git log**: Used to check the history of the Git repository.
**Git branch**: Used to create one or multiple branches in the repository.
**Git merge**: Used to merge commits between two branches.
**Git Rebase**: Merges changes from two branches, making the git history linear.
**Git Tag**: Used to create a tag for a specific state in the repository.
**Git Push**: Pushes commits from the local to the remote repository.
**Git pull**: Fetches and merges commits from remote to local repository.

# 3. Packaging, Release, and Continuous Integration

Unit 3.1 - Introduction to Jenkins

Unit 3.2 - Continuous Integration and Delivery with Jenkins

## Key Learning Outcomes

**At the end of this module, you will be able to:**

1. Explain the Jenkins and it set up.
2. Explain the different plugins for Jenkins.
3. Explain how to integrate Jenkins with GitHub.
4. Explain the need for continuous integration.
5. Explain how to build the code using GNU.
6. Explain how to run automated testing and reporting.

# UNIT 3.1: Introduction to Jenkins

## Unit Objectives

**At the end of this unit, you will be able to:**

1. Explain why we need Jenkins.
2. Illustrate how to setup Jenkins on a pc.
3. Explain how to manage different plugins in Jenkins.
4. Explain how to integrate Jenkins with GitHub.

## 3.1.1 Introduction to Continuous Integration

Continuous Integration (CI) is a software development practice in which developers integrate code into a shared remote repository. The developers usually perform multiple integrations several times a day. It helps developers detect errors quickly during the initial stage itself. Traditionally, integration is performed in preparation for the deployment of a new release. CI doesn't get rid of the bugs and errors, but it does help the developers reduce the bugs in the code. Usually, both testing and building are automated, so that every commit to the central repository triggers an automatic build. Most modern apps nowadays use CI as a basic approach to integrate their source code with version control systems and test it in parallel to validate it. In recent years, CI has become one of the best practices in software development.

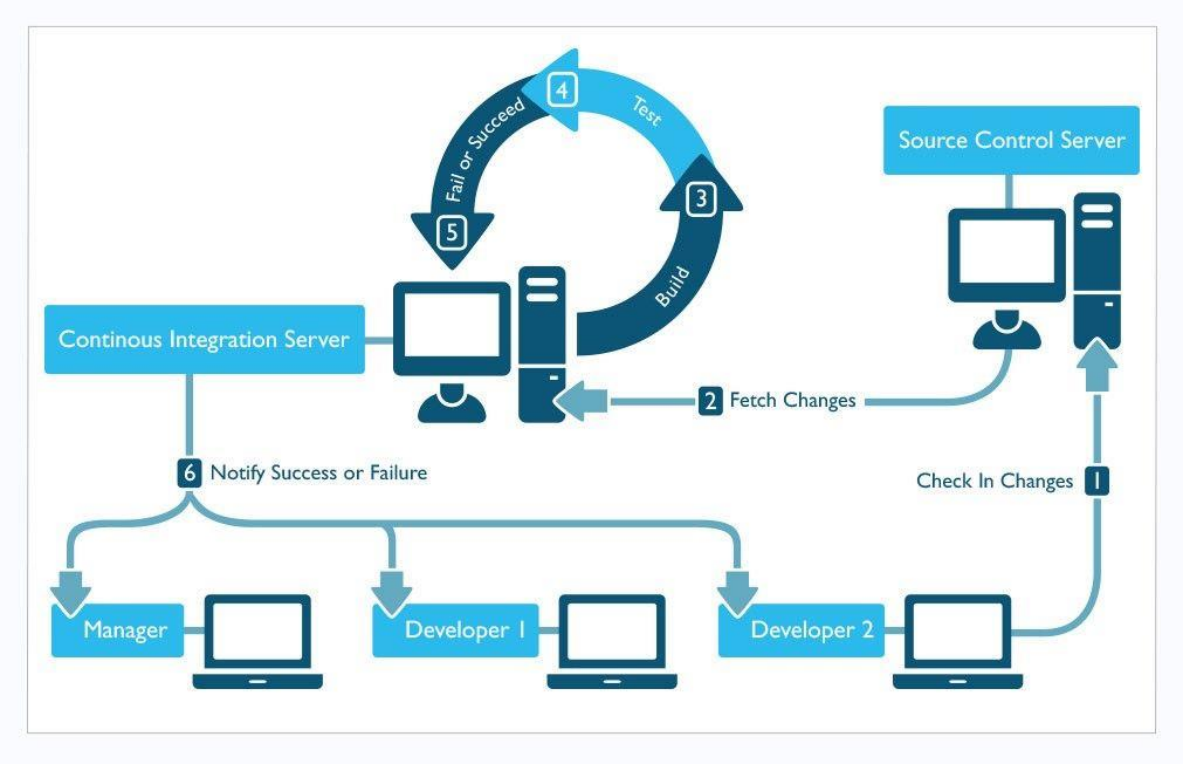



*Fig 3.1.1 Continuous Integration*

## 3.1.1.1 Continuous Integration Principles

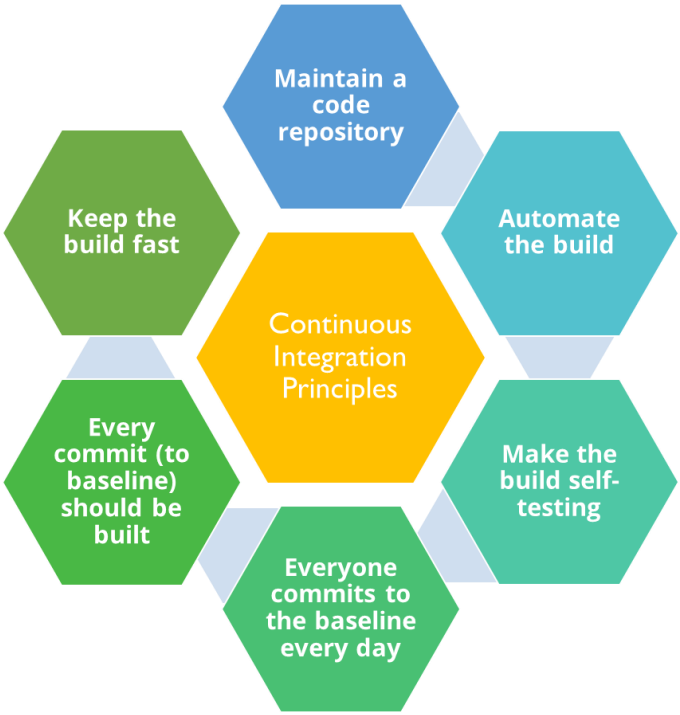



*Fig 3.1.1.1 Continuous Integration Principles*

**Maintain a code repository:** This practice advocates the use of a revision control system for the project's source code. All artifacts required to build the project should be placed in the repository.

**Automate the build:** One single command should have the ability to build the system. Many building tools, such as "Make", have been around for many years. Other, more recent tools are mostly used in continuous integration environments.

**Make the build self-testing:** Once the code is built, tests are run on the codes to confirm that they behave as expected by the developers.

**Everyone commits to the baseline every day:** By committing regularly, every committer can reduce the number of conflicting changes. Checking in a week's worth of work is risky, as it may conflict with other features and be very difficult to resolve.

**Keep the build fast:** The build needs to be completed quickly so that if there is a problem with integration, it is quickly identified.

**Every commitment (to a baseline) should be built as follows:** The system should build commits to the current working version to confirm that they integrate correctly. Commonly, "automated continuous integration" is used, although this may be done manually.

**Everyone can see the results of the latest build:** It should be easy to find out if the build breaks, and if so, it should be easy to identify who made the relevant change and what that change was.

## 3.1.1.2 Why Continuous Integration?

Integration bugs are traced early and are easy to track down due to small change sets. By this method, both time and money are saved over the lifespan of a project. CI avoids last-minute chaos at release dates when everyone tries to check in their slightly incompatible versions. When unit tests fail or a bug appears, reverting the codebase to a bug-free environment without debugging loses only a small number of changes. Frequent code check-ins encourage developers to write more modular, less complex code.

## 3.1.2 Introduction to Continuous Deployment

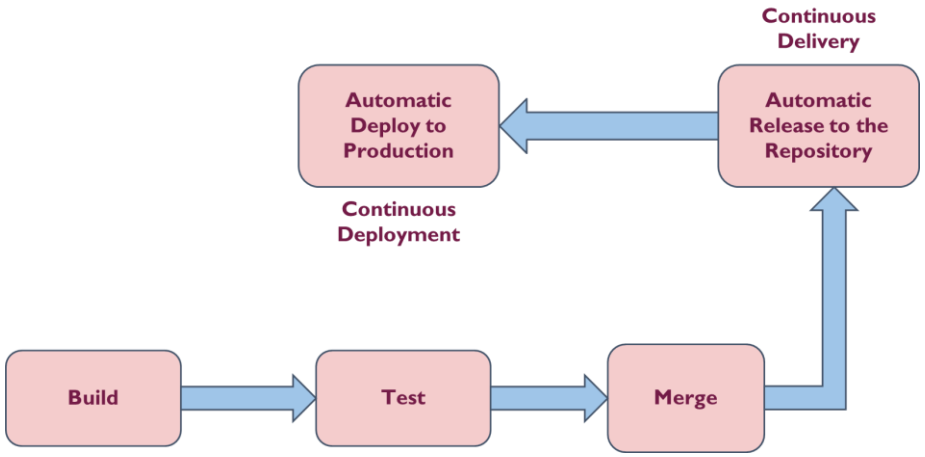



*Fig 3.1.2 Continuous Deployment*

Continuous Deployment (CD) is also known as an extension of Continuous Integration (CI). CD helps developers deliver features using an automated deployment procedure. In this process, any commit that passes automated testing automatically gets deployed in the environment. CD helps generate feedback immediately after source code commits.

## 3.1.2.1 Why Continuous Deployment?


*Fig 3.1.2.1 Continuous Deployment*

**Faster development:** CD developers don't have to waste a lot of time performing manual tasks. As soon as development is done, code gets integrated with the live environment.
**High-quality source code:** With CI and CD implementation, bug fixing is a breeze because we have a plethora of test cases to choose from. With every commit, the complete flow will be tested.
**Small changes as part of the releases:** With CI and CD, every time deployment is happening, only a few changes are getting deployed.
**Flexibility:** Performing changes with a CD is very flexible for developers since they can easily release those changes to the live environment.

## 3.1.3 Introduction to Jenkins

Jenkins is one of the most famous open-source continuous integration and continuous delivery servers available today. Initially, it was called Hudson and was developed at Sun Microsystems in 2004–2005. Later, it was forked from Hudson and renamed Jenkins in 2011, as the result of a dispute between the Hudson community and Oracle. The creator of Hudson or Jenkins, Kohsuke Kawaguchi, became the chief technical officer for Cloudbees in 2014, and Cloudbees now commercially offers Jenkins as a cloud solution. Jenkins provides all the features and functionalities that you require to build a robust continuous integration and continuous delivery pipeline.

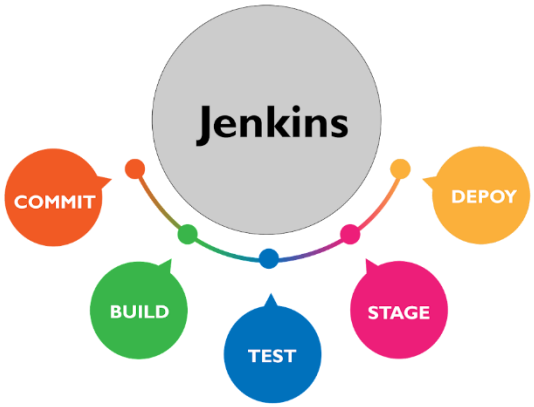



*Fig 3.1.3 Jenkins*

## 3.1.3.1 Jenkins Architecture

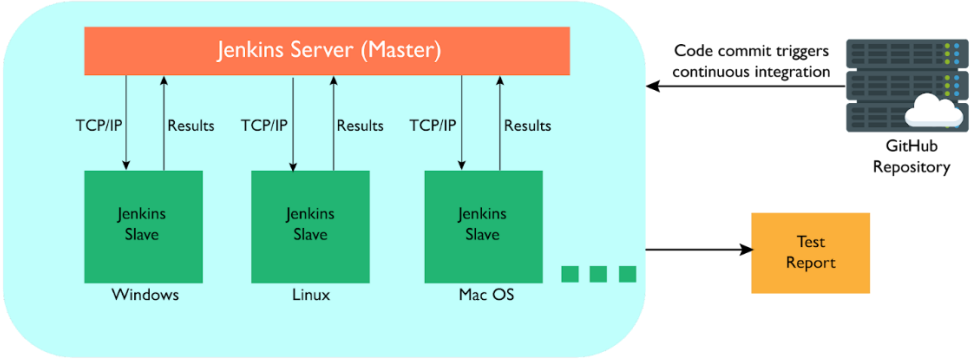



*Fig 3.1.3.1 Jenkins Architecture*

## 3.1.4 Jenkins Installation and Configuration

### 3.1.4.1 Jenkins Installation

Jenkins can be installed on a variety of operating systems and cloud platforms. Use the below link to access the installation steps for various platforms. https://jenkins.io/download/ . Once you open the above link, please select the "Ubuntu/Debian" option to get the installation procedure and the package details for Ubuntu. Now, login to the "Ubuntu Linux" operating system and login with your root ID so that we can proceed with the installation. For Windows-based platforms, we can download an "exe" file from Jenkins' portal. https://jenkins.io/download/thank-you-downloading-windows-installer-stable/ .

### 3.1.4.2 Jenkins Linux Installation

Before moving forward with Jenkins installation, proceed with JDK installation on the Ubuntu server.

```
apt update
apt install openjdk-8-jdk
```

After installing JDK, run the commands below to install Jenkins.

- wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add -

Using the following command, we must add the Jenkins package URL to the apt source list:

- echo "deb https://pkg.jenkins.io/debian-stable binary/" > /etc/apt/sources.d/jenkins.list

Then execute the below commands to install the Jenkins package using the apt command:

```
apt-get update
apt-get install Jenkins
```

Once Jenkins is installed, open a browser on your local machine and enter the URL: x.x.x.x:8080

### 3.1.4.3 Jenkins Windows Installation

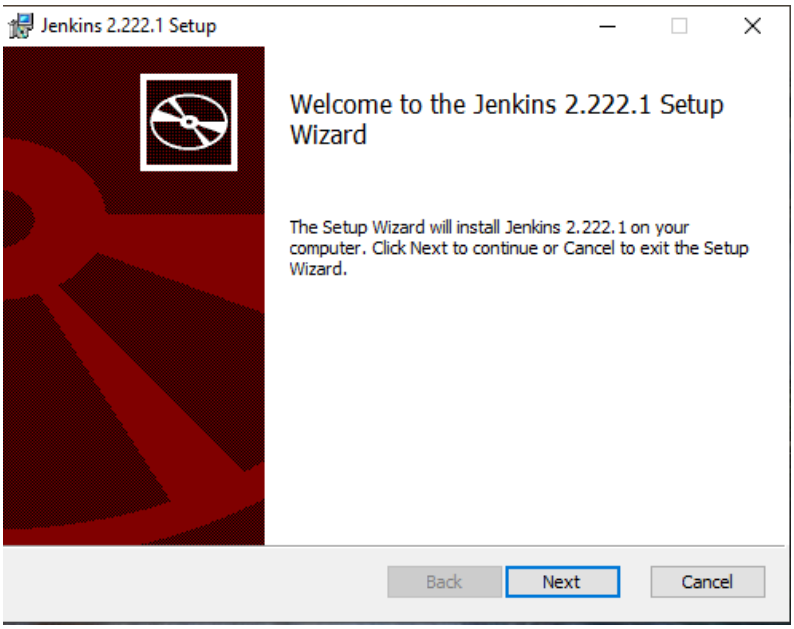



*Fig 3.1.4.3 Windows Installation*

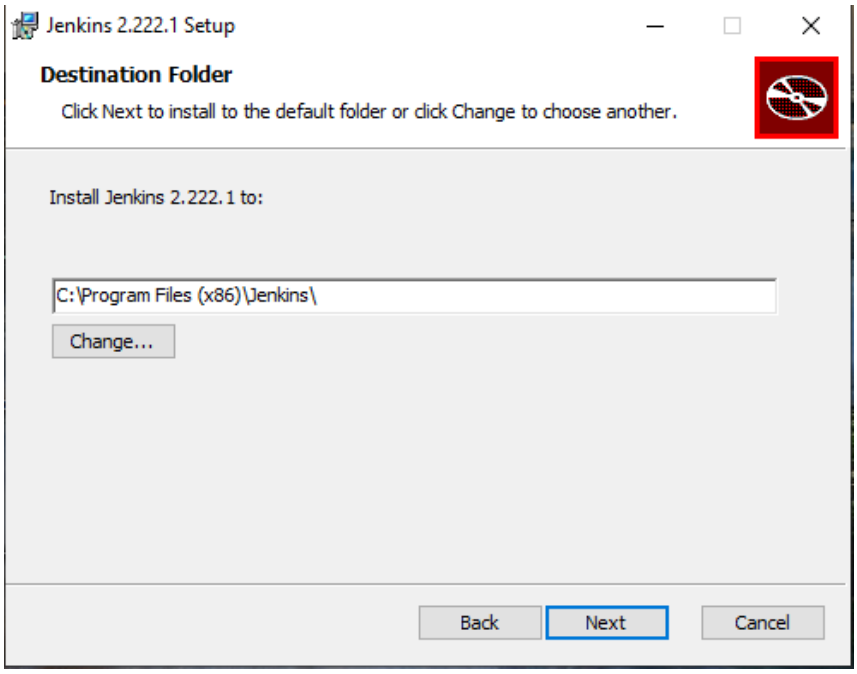



*Fig 3.1.4.3 Windows Installation*

## 3.1.5 Jenkins Jobs

Developers create the source code on their workstations and typically compile it only locally. However, if the developer is unavailable and we need to perform any build activity during the off hours, this is extremely difficult to accomplish. To get rid of these limitations, we can use CI/CD tools that can be scheduled, built, and deployed automatically. Using these tools, we can create separate jobs for each project that will execute all steps in a sequence automatically.

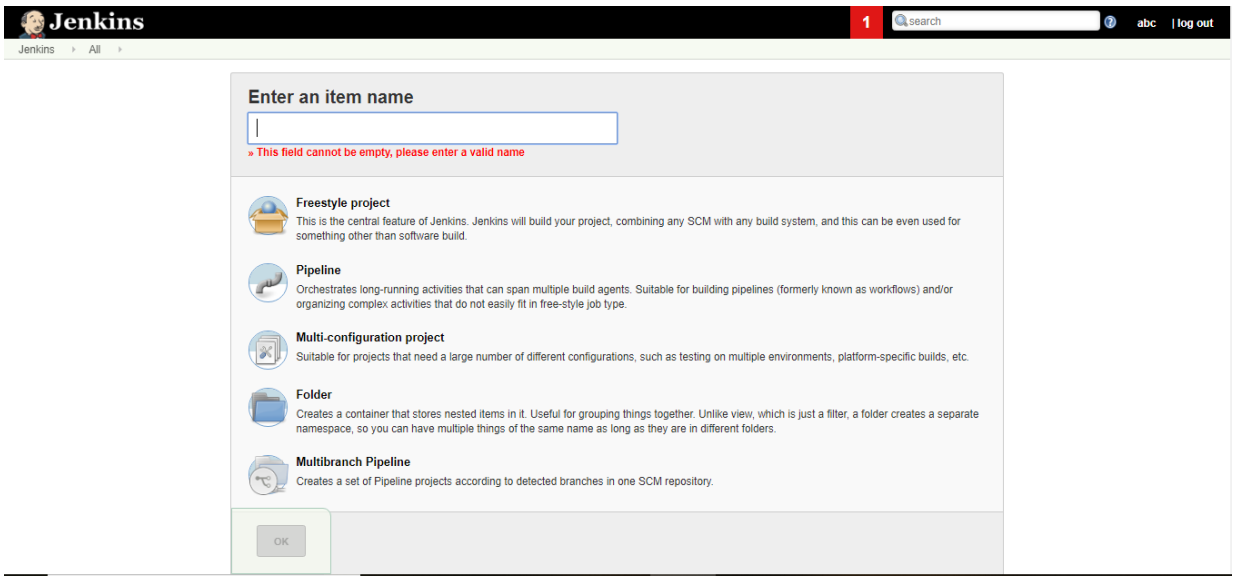



*Fig 3.1.5 Jenkins jobs*

# 3.1.6 Jenkins Build and Build Triggers

## 3.1.6.1 Jenkins Build

The process of compiling and packaging code into executable binaries is known as build. Build tools are scripts that automate the preparation of executable applications from the source code. In smaller projects, usually developers perform all build activities, but in larger projects, developers need build scripts for build automation. Developers can store these build scripts with source code inside the version control system.

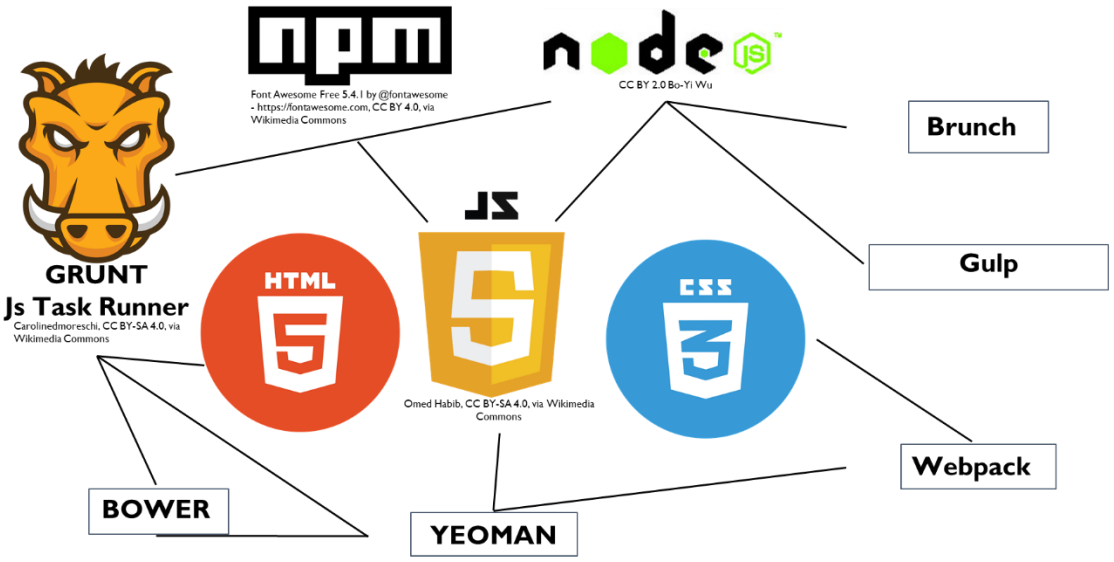



*Fig 3.1.6.1 Jenkins Build*

## 3.1.6.2 Jenkins Build Triggers

While configuring Jenkins jobs, we have some triggers that can be configured. We have the following triggers in Jenkins:

**Trigger builds remotely (e.g., from scripts):** With this, we can run Jenkins builds remotely using shell commands and the REST API.

**Build after other projects are built:** This will be used to configure the upstream Jenkins job in the original job.

**Build periodically:** This will be used to schedule Jenkins jobs using crontab-like syntax.

**GitHub hook trigger for GITScm polling:** This will be used to configure webhooks, which will help us trigger automated builds.

**Poll SCM:** SCM polling will help us to perform polling on the version control system so that, once any code is changed, Jenkins builds will be triggered automatically.

## 3.1.7 Schedule Jobs in Jenkins

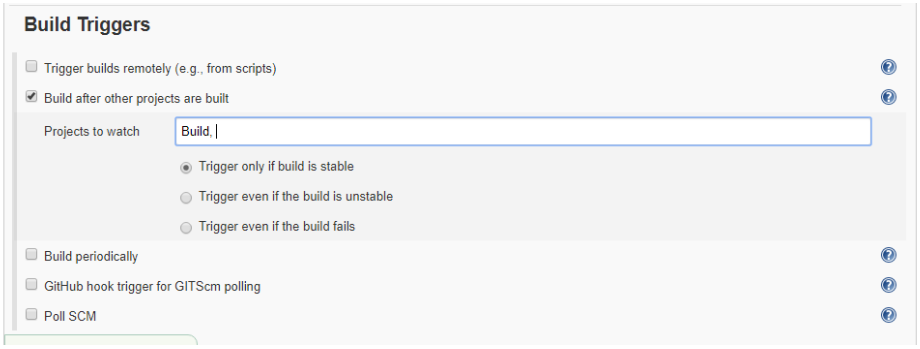



*Fig 3.1.7 Schedule Jobs in Jenkins*

## 3.1.8 Jenkins Post Build Actions

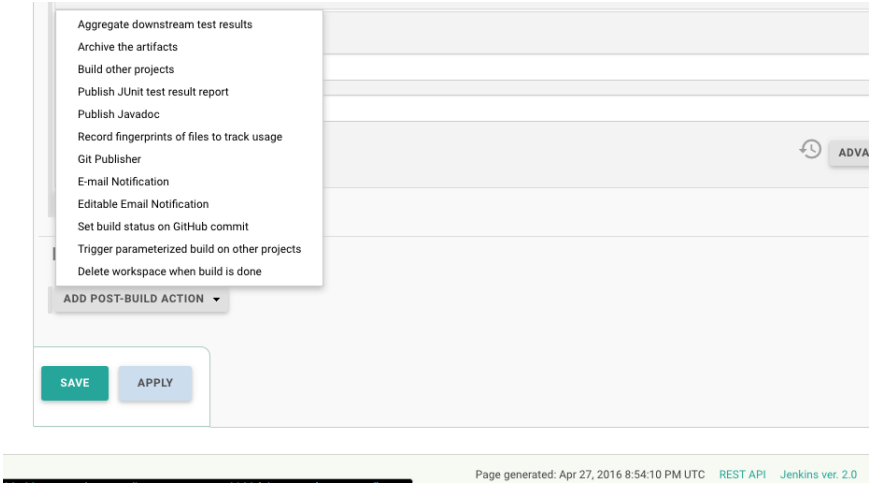



*Fig 3.1.8 Jenkins Post Build Actions*

## 3.1.9 Introduction to Build, Build Automation, and Distributed Build

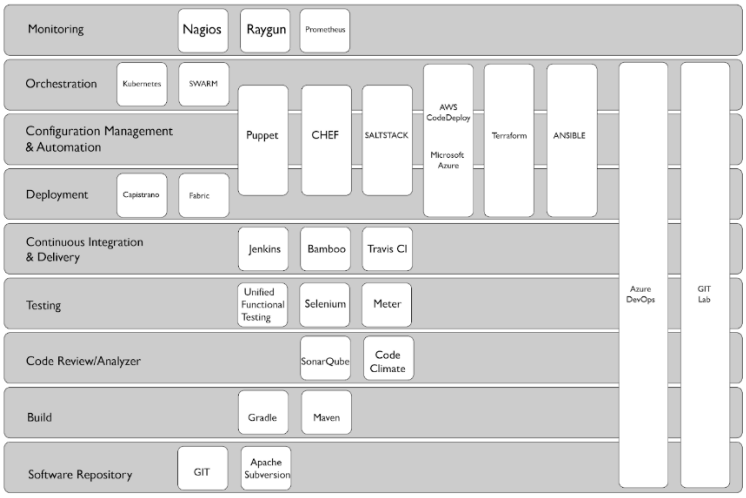



*Fig 3.1.9 DevOps Tools (Products)*

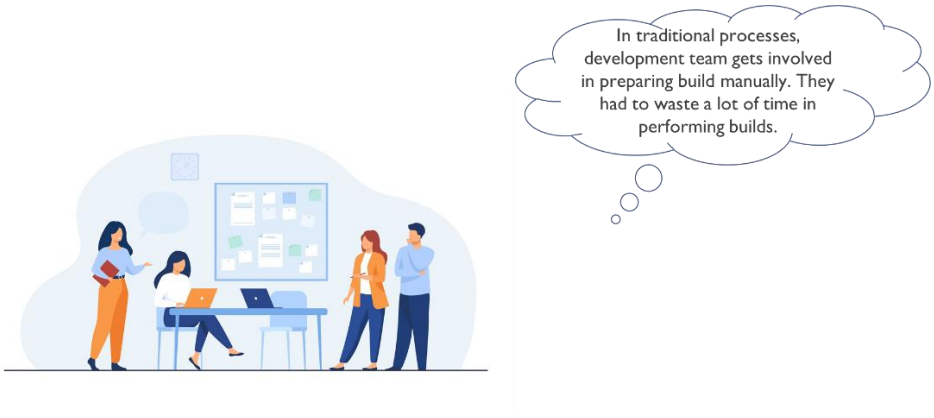



*Fig 3.1.9 Problems with Manual Build Process*

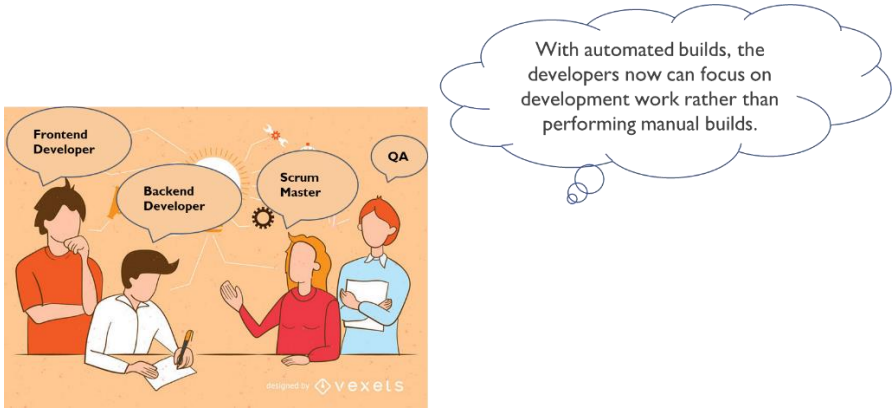



*Fig 3.1.9 After Build Automation*

## 3.1.9.1 Distributed Builds

Jenkins' distributed build feature distributes build load across multiple machines. Instead of running builds on a single master node, we can add slaves to Jenkins to build these heavy projects. A master node can distribute the load across multiple slave machines. If one slave machine fails, another will take over the build and execute it.

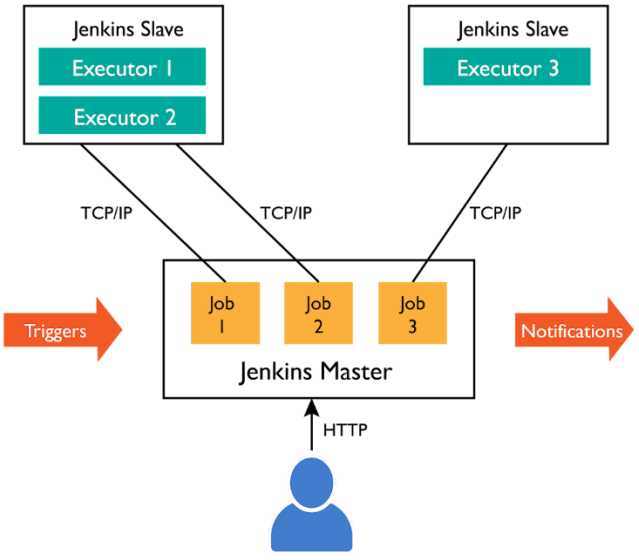



*Fig 3.1.9.1 Distributed Builds*

## 3.1.9.2 Dynamic Jenkins Slave Agents

Jenkins supports various cloud plugins, using which we can create replaceable agents. We do not need to set up any slave machines for the master node. Whenever build is initialized, a node machine will be created, and once build is completed, a node machine can be stopped or terminated. These agents will help reduce the expense by creating agents whenever they are required and terminating them once the work is done. For example, we can use the EC2 plugin, Docker plugin, and Kubernetes plugin to setup replaceable agents for running distributed builds.

## 3.1.9.3 Jenkins Master Slave Communication

The master node establishes connections to the agent slave machine to send build requests. The agent slave machine will be completely controlled by the master node. The master node needs to have network access to the slave machine; if the network is not available, the slave machine can be started via the Java Network Launch Protocol (JNLP). The following methods can be used to connect a master and slave:

- Secure Shell (SSH) agent launch
- Windows agent launch
- JNLP agent launch
- Agent jar file launch

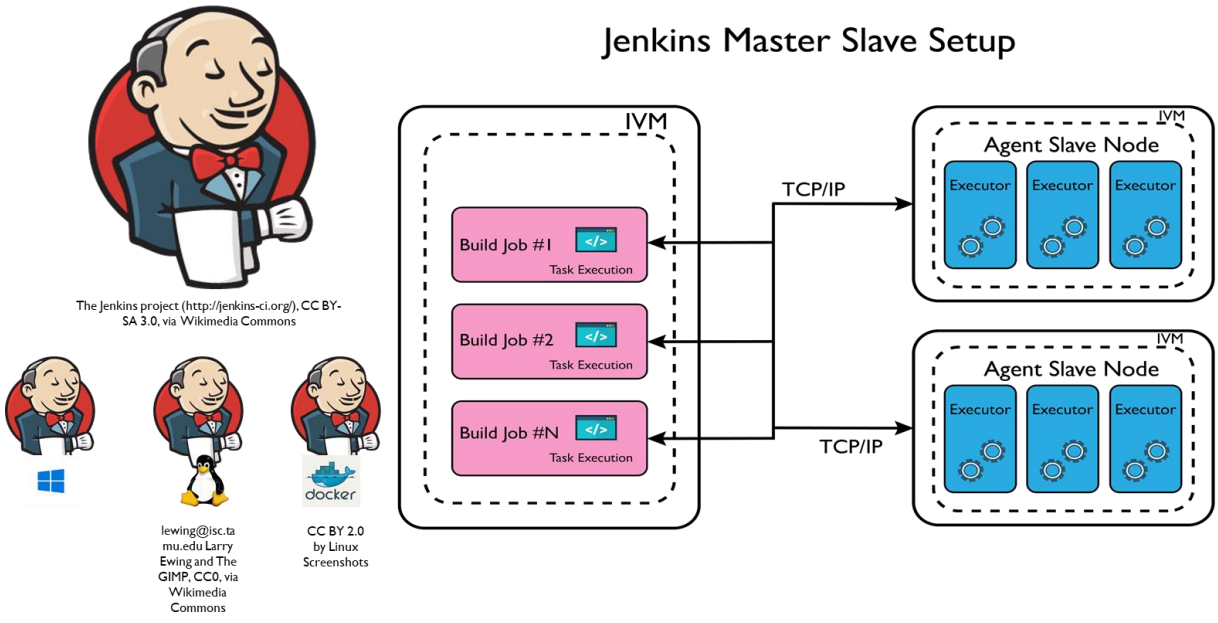



*Fig 3.1.9.3 Jenkins Master Slave Communication*

## 3.1.10 Jenkins Plugins

Plugins help enhance the functionalities of the Jenkins application as per user-specific requirements. There are more than a thousand plugins available that can be installed on the Jenkins master node. These plugins are also available as open source, which we can easily integrate within Jenkins.

**Top Jenkins Plugins:**

Below are top 8 plugins which is required for any project:

- Pipeline
- Maven
- Amazon EC2
- HTML Publisher
- Post Build Task
- Parameterized Trigger
- Throttle Builds
- Green Balls

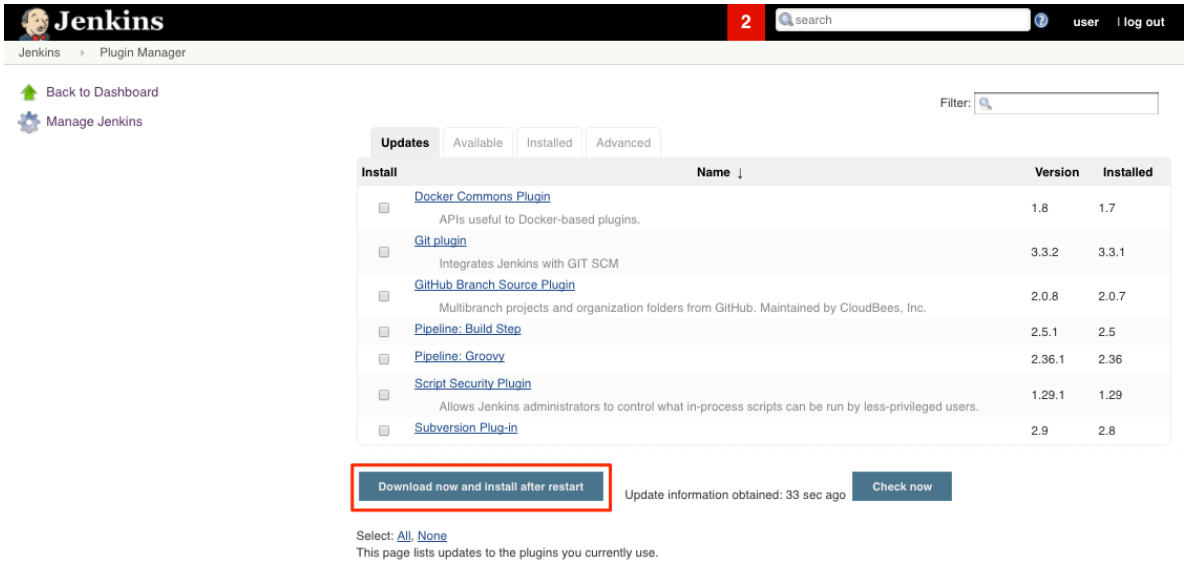



*Fig 3.1.10 Jenkins Plugins*

## 3.1.11 Jenkins Authentication and Authorization

**Jenkins Authentication:**

We have three important settings in Jenkins to manage security:

**Enable security:** If we disable this option, Jenkins will be open to everyone, and we will not be getting any security benefits from that.

**Security Realm (Authentication):** It will help us to select the authentication source that will be used by Jenkins to authenticate the users.

**Authorization:** In this way, we can assign some roles to the individual users without giving similar admin access to every user.


*Fig 3.1.11 Jenkins Authentication*

**Jenkins Authorization**

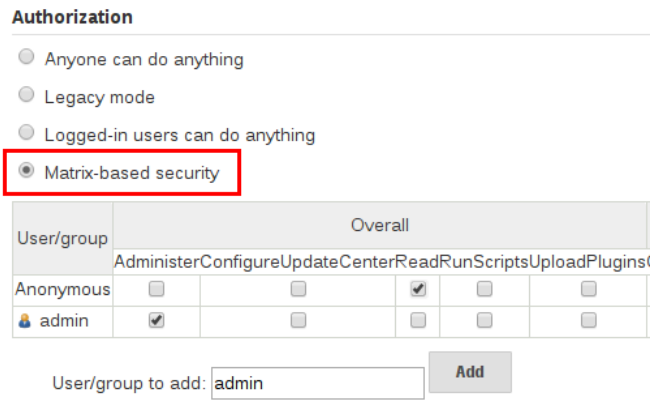



*Fig 3.1.11 Jenkins Authorization*

## 3.1.12 Parameterizing the Build

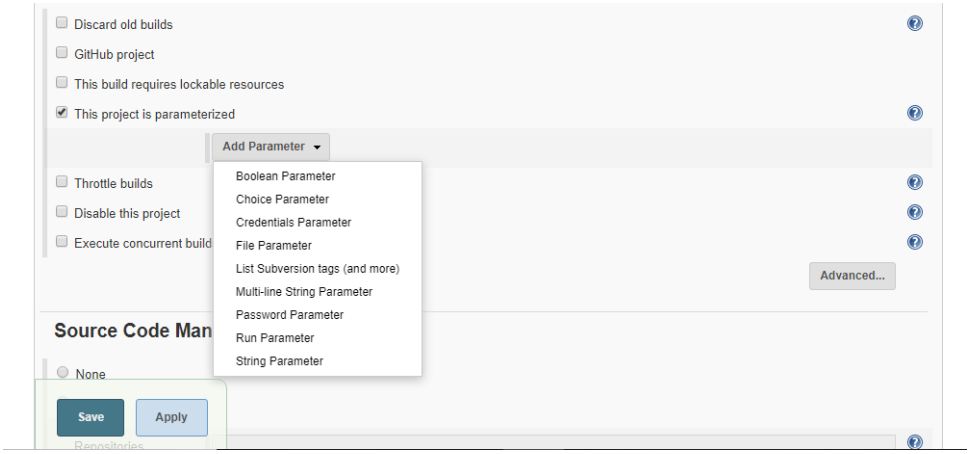



*Fig 3.1.12 Jenkins Parameters*

While working with Jenkins, we can provide parameters that help us trigger jobs with multiple parameters. We can use various types of build parameters in both freestyle and pipeline jobs. While triggering the job, we can select a parameter's value, and depending on that, those values will be processed by Jenkins' job. With reference to the image in the previous slide, we can see multiple parameter types that we can configure in Jenkins jobs.

# 3.1.13 Jenkins Integration with Other Tools

## 3.1.13.1 Integrate Jenkins with Ant

The Ant build tool can be used to trigger builds and test case execution automatically without manual effort. While configuring a job, in the build option, select "Invoke Ant" and choose the "Ant" version. You can configure the "build.xml" file as the build script, but in the case of a different build script, you can specify it in the "Build File" parameter.

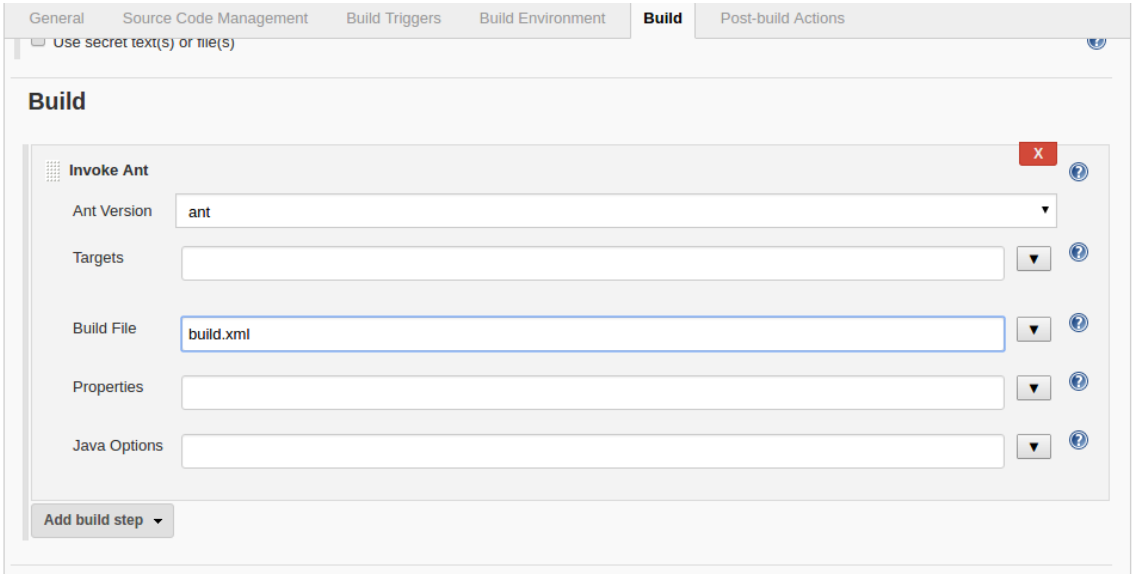



*Fig 3.1.13.1 Integrate Jenkins with Ant*

## 3.1.13.2 Integrate Jenkins with Maven

Maven is a project and build management tool used to compile, package, and deploy artifacts to an artifactory. Maven integration can be installed within Jenkins to support Maven build automation using Jenkins. We need to move to the Build section and select the "Invoke top-level Maven" target option from the dropdown.

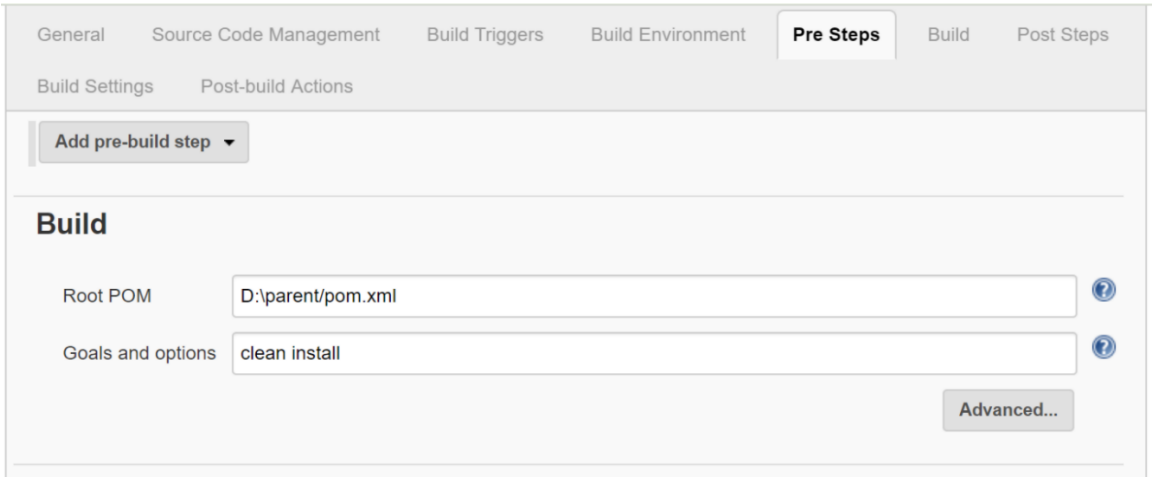



*Fig 3.1.13.2 Integrate Jenkins with Maven*

## 3.1.13.3 Integrate Jenkins with Shell Scripts

Jenkins supports most of the build automation tools like Ant, Maven, Gradle, MSBUILD, and others, but some applications may require a different setup. We can use Jenkins' Shell Script plugin, which allows a user to execute any shell script for Jenkins automation. In order to configure Shell Script, select "execute shell" in the build step. Finally, enter the name of your shell script and click "save," then run "build."

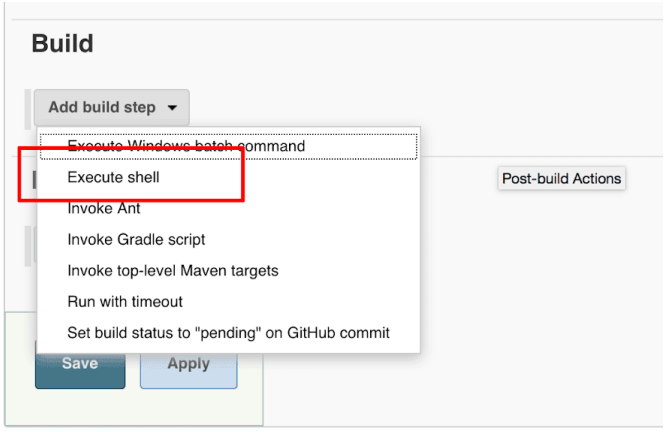



*Fig 3.1.13.3 Integrate Jenkins with Shell Scripts*

## 3.1.13.4 Integrate Jenkins with Python Scripts

Some automation may not be always possible with Maven, Ant, and other build tools. We may need some generic approach for automating processes where shell scripts and Python scripts can help users automate their manual efforts. We can use the "Execute Shell" option in the build step to configure the execution of Python scripts.

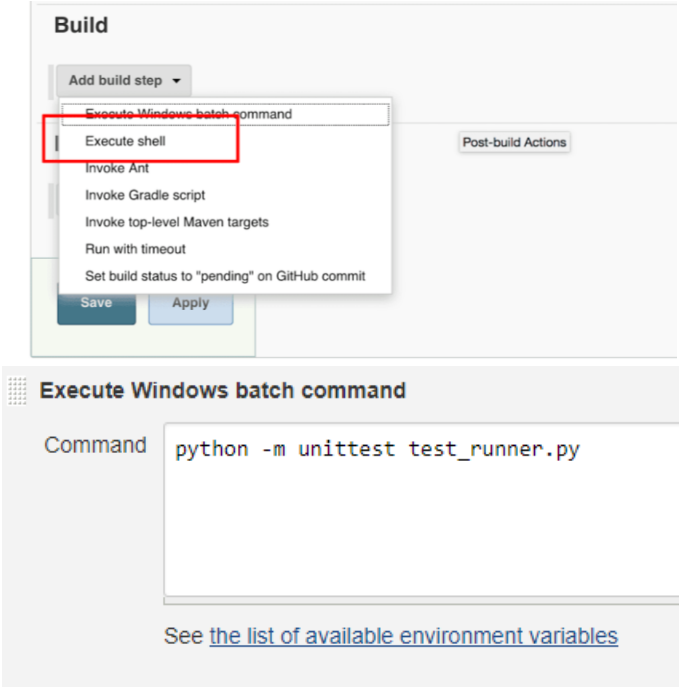



*Fig 3.1.13.4 Integrate Jenkins with Python Scripts*

# UNIT 3.2: Continuous Integration and Delivery with Jenkins

## Unit Objectives

**At the end of this unit, you will be able to:**

1. Explain the need for continuous integration.
2. Explain how to build the code using GNU.
3. Illustrate how to build the various packages.
4. Explain how to run automated testing and reporting.
5. Illustrate how to publish and deliver the artifacts.

## 3.2.1 What is DevOps?

DevOps is the practice of operations and development engineers participating together in the entire service lifecycle, from design through the development process to production support.

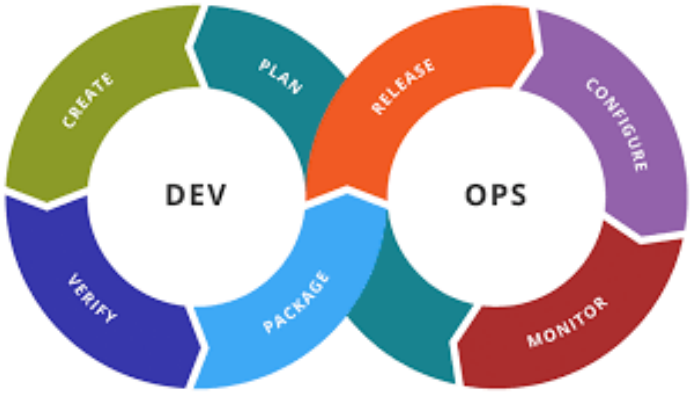



*Fig 3.2.1 DevOps*

**What exactly do DevOps do?**

DevOps is all about the unification and automation of processes, and DevOps engineers are instrumental in combining code, application maintenance, and application management. All these tasks rely on understanding not only development life cycles but also DevOps culture, including its philosophy, practices, and tools.

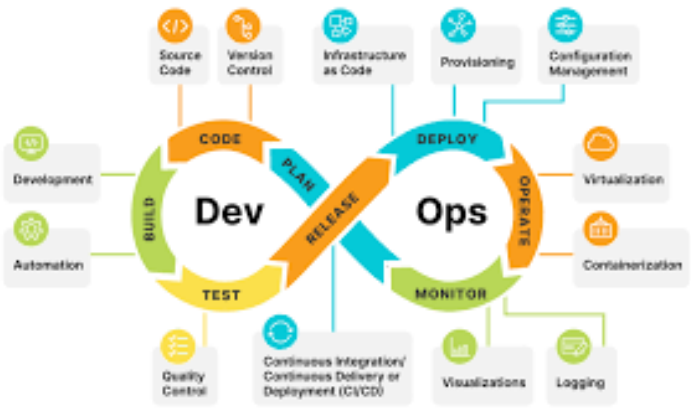



*Fig 3.2.1 Development Lifecycle*

## 3.2.1.1 DevOps Workflow

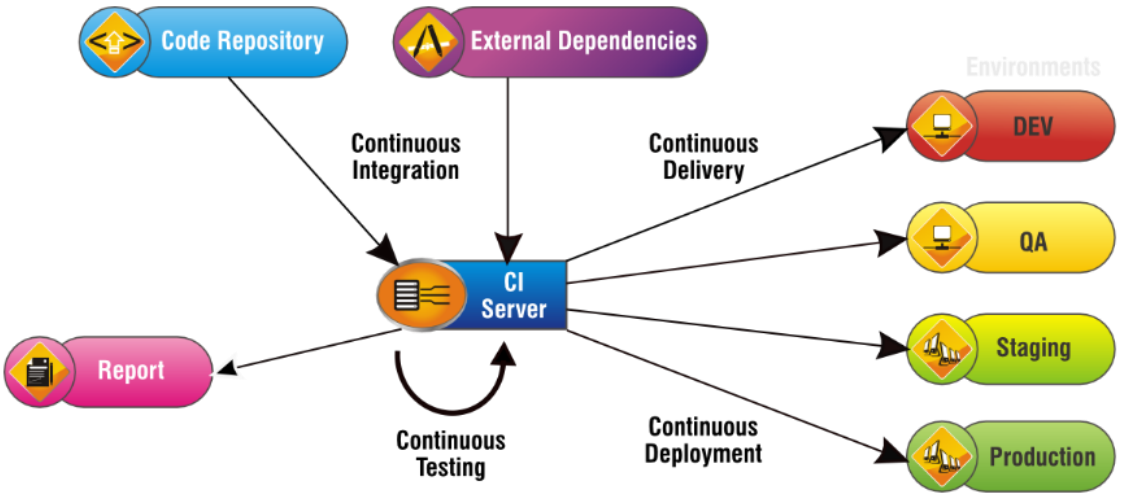



*Fig 3.2.1.1 DevOps Workflow*

## 3.2.2 Basics of Virtualization

Virtualization is the process or technique that can create a virtual version of IT datacenter components such as compute, storage, network, etc. Hypervisor is one of the most used virtualization technologies to create virtual IT infrastructures. It is an important technology and the main driver of cloud-based infrastructure. Hypervisor is one of the most widely used by many cloud service providers to virtualize traditional hardware IT infrastructure and offer it as one or more cloud services.


*Fig 3.2.2 Virtualization Basics*

**Benefits of Virtualization:**

- Reduced upfront hardware costs and continuing operating costs.
- Minimized or eliminated downtime.
- Increased IT productivity and responsiveness.
- Greater business continuity and disaster recovery responses
- Simplified data center management.
- Faster provisioning of applications and resources.

## 3.2.3 Version Control with Git

**What is Version Control?**

A system that keeps records of changes. It allows for collaborative development. It allows you to know who made what changes and when. It allows you to revert any changes and go back to the previous state.

## 3.2.3.1 What is Git?

Git is a distributed version control system for tracking changes in any set of files, usually used for coordinating work among programmers collaboratively developing source code during software development.

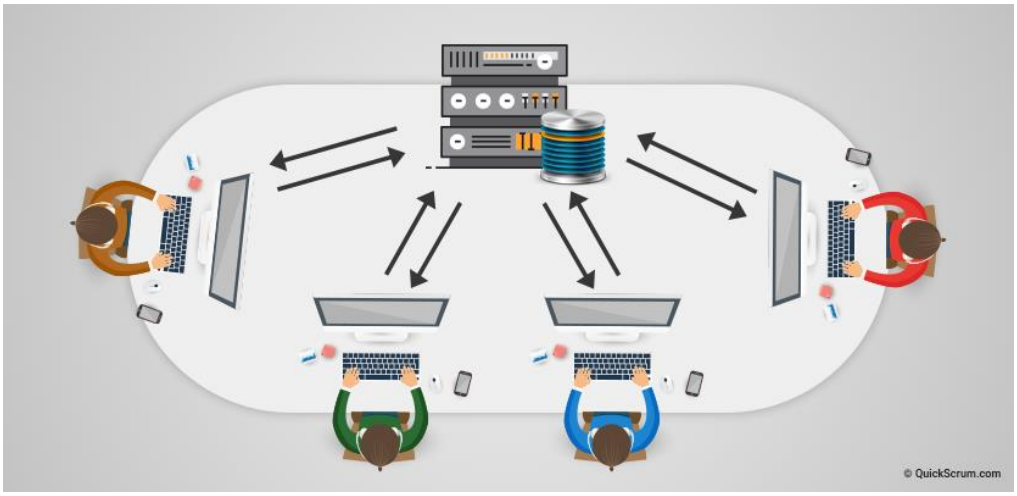



*Fig 3.2.3.1 Git*

**Basic Git Commands**

- git init --- Create a new local repository
- git clone /path/to/repository --- Check out a repository Create a working copy of a local repository:
- git clone username@host:/path/to/repository --- For a remote server, use:
- git add <filename> --- Add files Add one or more files to staging (index):
- git add * --- add all files for commit
- git commit -m "Commit message" Commit--- Commit changes to head (but not yet to the remote repository):
- git commit -a --- Commit any files you've added with git add, and commit any files you've changed since then:
- git push origin master --- Push Send changes to the master branch of your remote repository:
- git status --- Status List the files you've changed and those you still need to add or commit:

**Why use Git in your Organization?**

If your business depends heavily on it and software processes, or if you're a software development entity, Git radically changes the way your team will create and deliver work to you. Various processes, including design, development, product management, marketing, and customer support, can be easily handled and maintained using Git in your organization.

## 3.2.3.2 Feature Branch Workflow

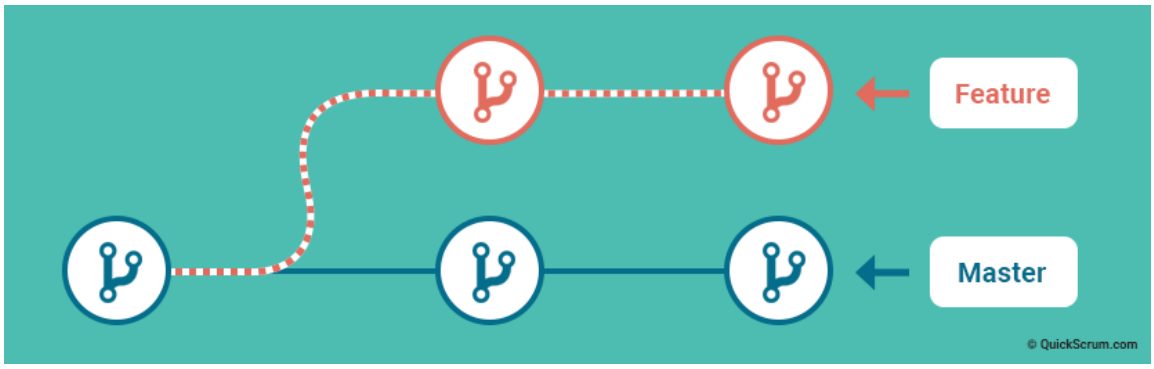



*Fig 3.2.3.2 Feature Branch Workflow*

Git has powerful branching capabilities. To start working, developers must first create a unique branch. Each branch functions in an isolated environment while changes are carried out in the codebase. This ensures that the master branch always supports production-quality code. Therefore, besides being more reliable, it's also much easier to edit code in a Git branch rather than editing it directly using an external editor.

## 3.2.3.3 Distributed Development

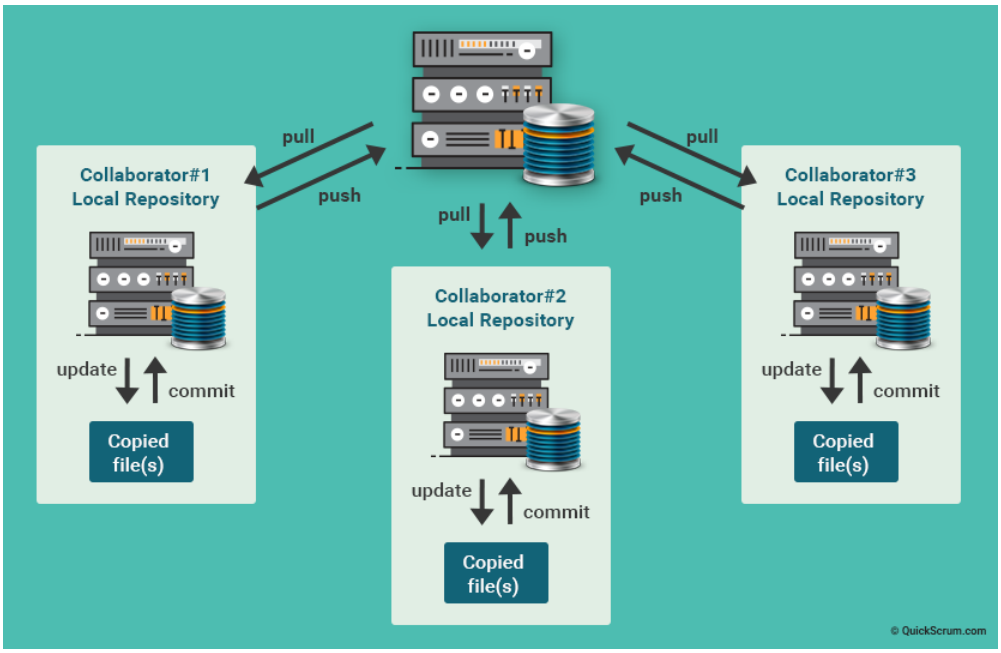



*Fig 3.2.3.3 Distributed Development*

Since Git is a distributed VCS, it offers a local repository to each developer with its own history of commits. Therefore, you don't require a network connection to create commits, inspect previous file versions, or check the differences between two or more commits. Also, it's much easier to scale the team. With Git, users can be easily added or removed as and when required. Other team members can continue working using their local repositories and are not dependent upon whether a new branch is added or an older one closed.

## 3.2.3.4 Community


*Fig 3.2.3.4 Community*

Git is very popular, widely used, and accepted as a standard version control system by most of the developer community. Using Git, it is much easier to leverage third-party libraries and encourage other developers to fork your open-source code. The sheer number of Git users makes it easy to resolve issues and seek outside help using online forums.

## 3.2.3.5 Pull Requests

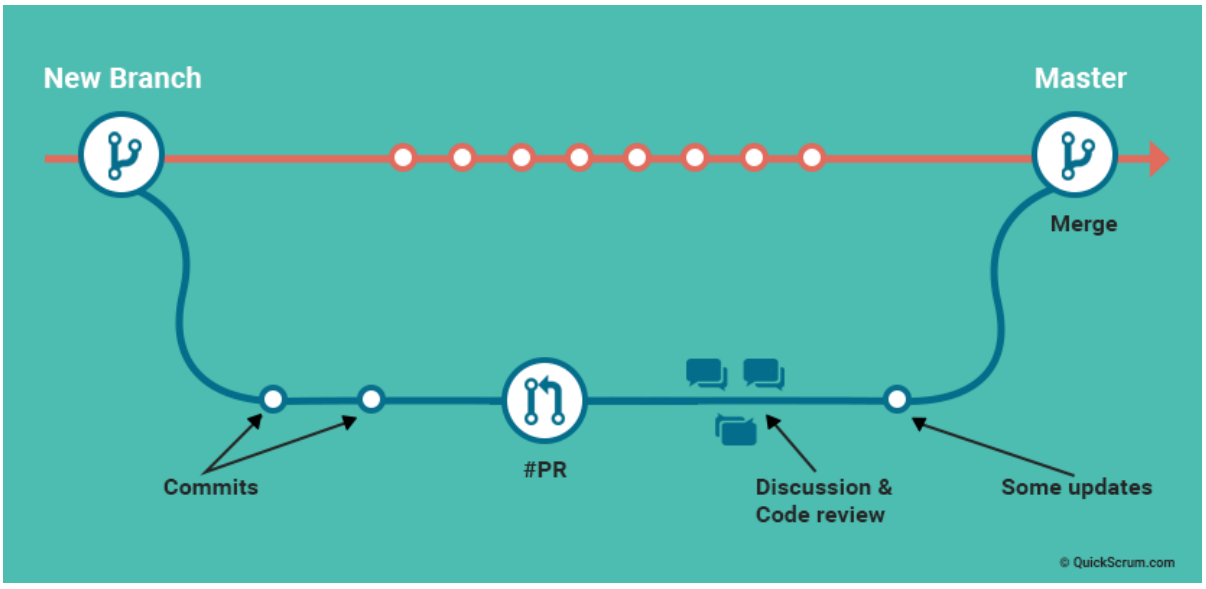



*Fig 3.2.3.5 Pull Requests*

A developer calls a pull request to ask another developer to merge one of his or her branches into the other's repository. Besides making it a lot easier for project leaders to monitor and track code changes, "pulling" also facilitates other developers discussing their work before integrating it with the codebase. Moreover, if a developer can't continue work owing to some hard technical problem, he or she can initiate a pull request to seek help from the rest of the team.

## 3.2.3.6 What is Distributed Control System?

A Distributed Version Control System (DVCS) brings a local copy of the complete repository to every team member's computer, so they can commit, branch, and merge locally. The server doesn't have to store a physical file for each branch; it just needs the differences between each commit.

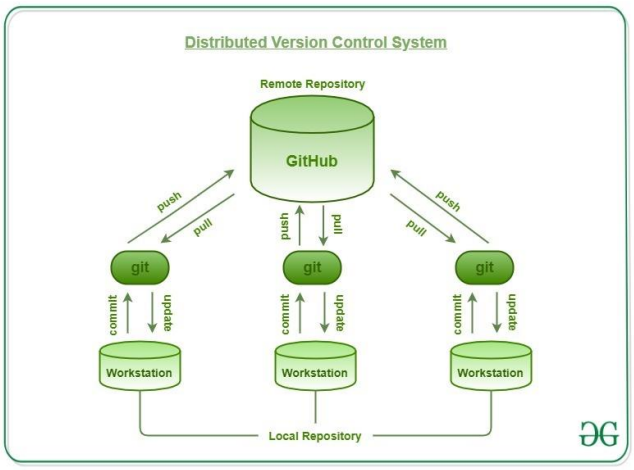



*Fig 3.2.3.6 Distributed Version Control System*

## 3.2.3.7 Differences Between Git and Github

**Git:** Git is a distributed version control system for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files. Its goals include speed, data integrity, and support for distributed, non-linear workflows.

**GitHub:** GitHub is a web-based Git repository hosting service that offers all the distributed revision control and Source Code Management (SCM) functionality of Git as well as adding its own features.

| S.No. | Git | GitHub |
|---|---|---|
| 1. | Git is a software. | GitHub is a service. |
| 2. | Git is a command-line tool | GitHub is a graphical user interface |
| 3. | Git is installed locally on the system | GitHub is hosted on the web |
| 4. | Git is maintained by linux. | GitHub is maintained by Microsoft. |
| 5. | Git is focused on version control and code sharing. | GitHub is focused on centralized source code hosting. |
| 6. | Git is a version control system to manage source code history. | GitHub is a hosting service for Git repositories. |
| 7. | Git was first released in 2005. | GitHub was launched in 2008. |


*Fig 3.2.3.7 Git and Github*

## 3.2.4 Software Build and Release Workflow

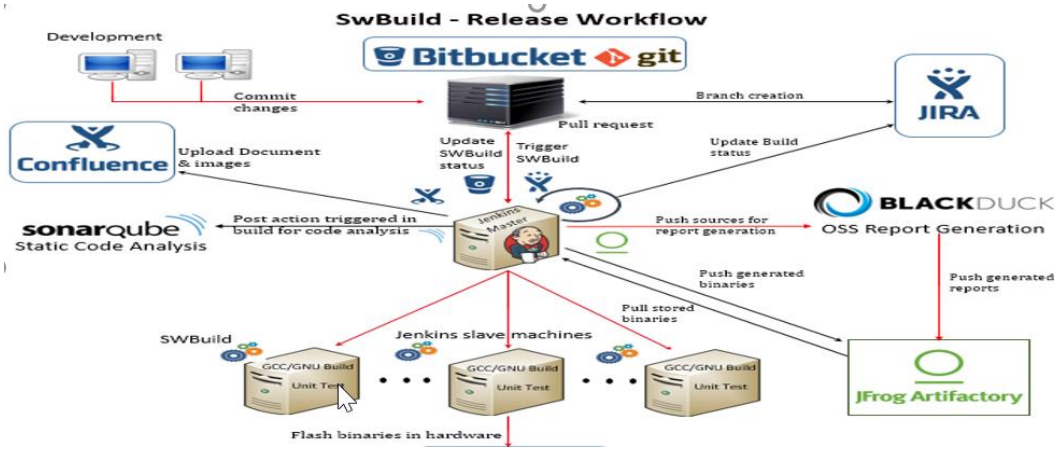



*Fig 3.2.4 Software Build and Release Workflow*

## 3.2.5 Overview of Jenkins

Jenkins is a Continuous Integration (CI) server or tool that is written in Java. It provides continuous integration services for software development, which can be started via the command line or a web application server. And it is good to know that Jenkins is free software to download and install.

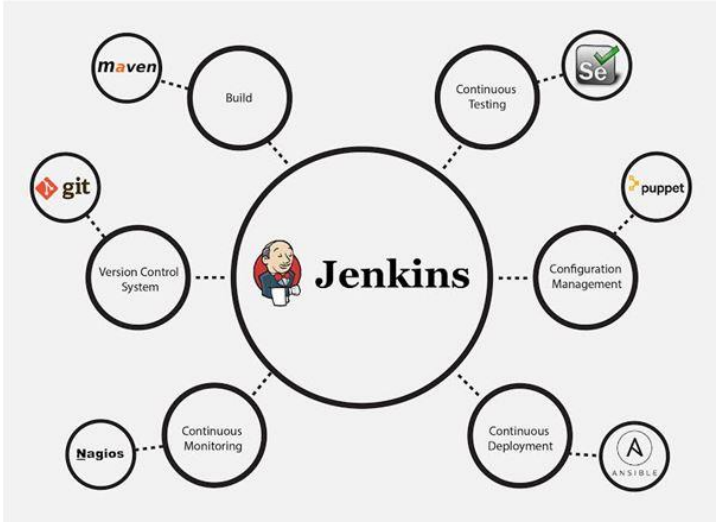



*Fig 3.2.5 Overview of Jenkins*

## 3.2.5.1 Types of Builds

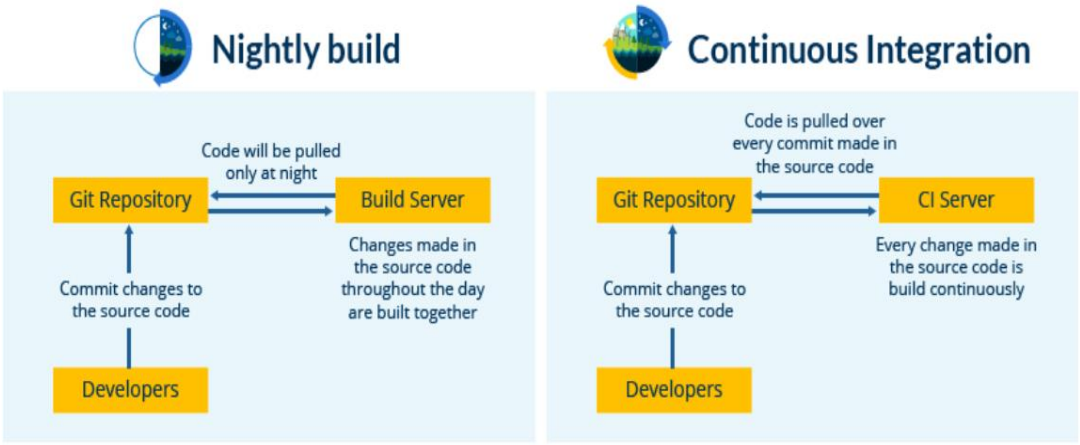



*Fig 3.2.5.1 Types of Builds*

## 3.2.5.2 Advantages of Jenkins

- It is an open-source tool with great community support.
- It is easy to install.
- It has more than 1,000 plugins to ease your work. If a plugin does not exist, you can code it and share it with the community.
- It is free of charge.
- It is built with Java, and hence, it is portable to all the major platforms.

## 3.2.5.3 Before and After Jenkins

| Before Jenkins | After Jenkins |
|---|---|
| The entire source code was built and then tested. Locating and fixing bugs in the event of build and test failure was difficult and time consuming, which in turn slows the software delivery process. | Every commit made in the source code is built and tested. So, instead of checking the entire source code developers only need to focus on a particular commit. This leads to frequent new software releases. |
| Developers have to wait for test results | Developers know the test result of every commit made in the source code on the run. |
| The whole process is manual | You only need to commit changes to the source code and Jenkins will automate the rest of the process for you. |

## 3.2.6 Continuous Integration

Continuous integration is a DevOps software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run. Continuous integration most often refers to the build or integration stage of the software release process and entails both an automation component (e.g., a CI or build service) and a cultural component (e.g., learning to integrate frequently). The key goals of continuous integration are to find and address bugs quicker, improve software quality, and reduce the time it takes to validate and release new software updates.

**Why is Continuous Integration Needed?**

In the past, developers on a team might work in isolation for an extended period of time and only merge their changes to the master branch once their work was complete. This made merging code changes difficult and time-consuming, and it also resulted in bugs accumulating for a long time without correction. These factors made it harder to deliver updates to customers quickly.

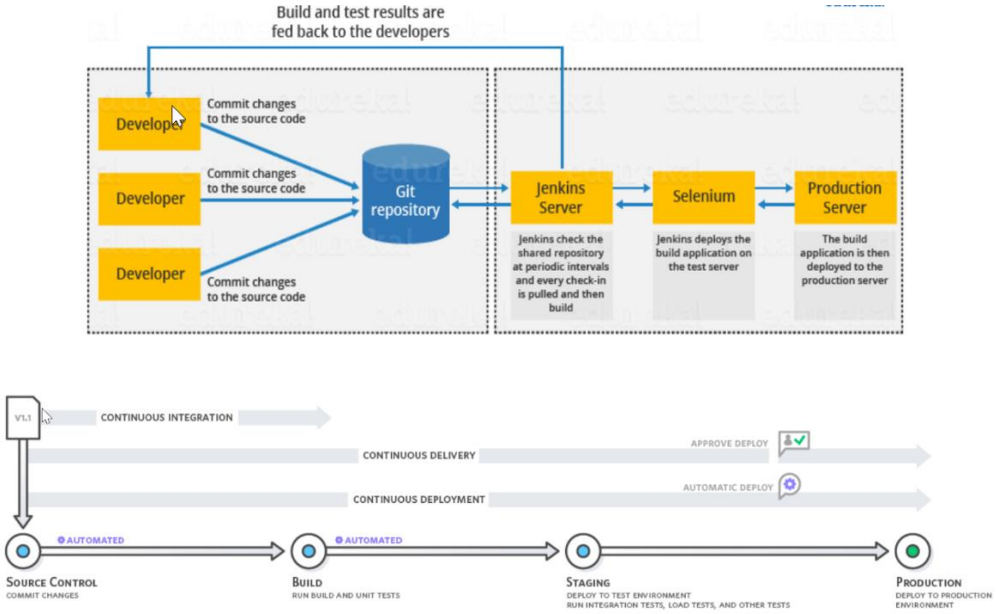



*Fig 3.2.6 Continuous Integration*

## 3.2.6.1 Continuous Integration Benefits

Improve Developer Productivity

Continuous integration helps your team be more productive by freeing developers from manual tasks and encouraging behaviors that help reduce the number of errors and bugs released to customers.

Find and Address Bugs Quicker

With more frequent testing, your team can discover and address bugs earlier before they grow into larger problems later.

Deliver Updates Faster

Continuous integration helps your team deliver updates to their customers faster and more frequently.

*Fig 3.2.6.1 Continuous Integration Benefits*

## 3.2.7 Continuous Delivery

Continuous delivery is a software development practice where code changes are automatically prepared for a release to production. A pillar of modern application development, continuous delivery expands upon continuous integration by deploying all code changes to a testing environment and/or a production environment after the build stage. When properly implemented, developers will always have a deployment-ready build artifact that has passed through a standardized test process.

**Why is Continuous Delivery Needed?**

Continuous delivery lets developers automate testing beyond just unit tests so they can verify application updates across multiple dimensions before deploying to customers. These tests may include UI testing, load testing, integration testing, API reliability testing, etc. This helps developers more thoroughly validate updates and pre-emptively discover issues. With the cloud, it is easy and cost-effective to automate the creation and replication of multiple environments for testing, which was previously difficult to do on-premises.

## 3.2.7.1 Benefits of Continuous Delivery

Perhaps the biggest advantage of continuous delivery is an increase in DevOps ROI. But there are other important continuous delivery benefits, too:

- Streamlining workflows.
- Lowering staffing costs.
- Improving operational confidence.
- Enhancing teamwork.

**Automate the Software Release Process**

Continuous delivery lets your team automatically build, test, and prepare code changes for release to production so that your software delivery is more efficient and rapid.

**Improve Developer Productivity**

These practices help your team be more productive by freeing developers from manual tasks and encouraging behaviors that help reduce the number of errors and bugs deployed to customers.

**Find and Address Bugs Quicker**

Your team can discover and address bugs earlier before they grow into larger problems later with more frequent and comprehensive testing. Continuous delivery lets you more easily perform additional types of tests on your code because the entire process has been automated.

**Deliver Updates Faster**

Continuous delivery helps your team deliver updates to customers faster and more frequently. When continuous delivery is implemented properly, you will always have a deployment-ready build artifact that has passed through a standardized test process.

*Fig 3.2.7.1 Benefits of Continuous Delivery*

## 3.2.8 Business Benefits of DevOps


*Fig 3.2.8 Business Benefits of DevOps*

## 3.2.9 Practical

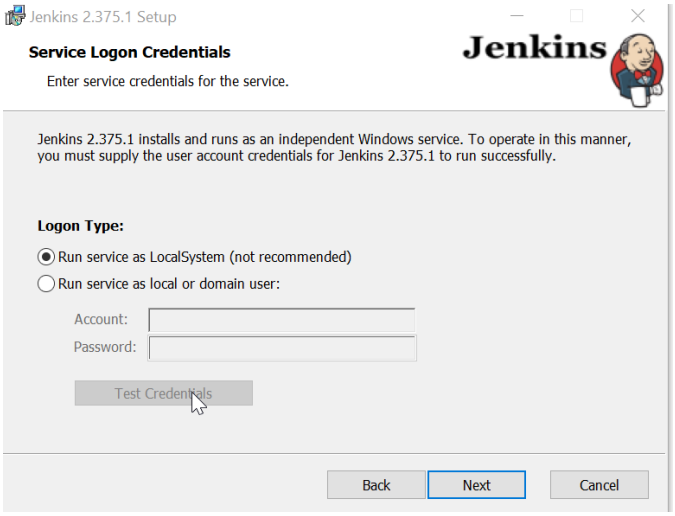
### 3.2.9.1 Tools Required for this Assignment

- Jenkins https://www.jenkins.io/download/thank-you-downloading-windows-installer-stable/
- cmoka https://cmocka.org/files/1.1/
- Cpplint pip3 install cpplint
- mingw https://sourceforge.net/projects/mingw/
- Git https://git-scm.com/downloads

### 3.2.9.2 Jenkins Installation

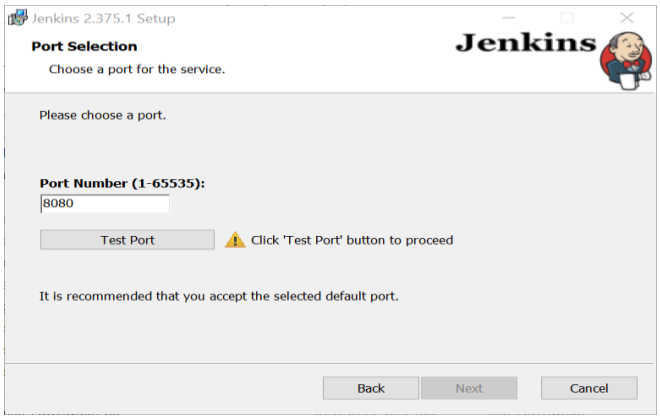
**Step 1:** Download jenkins.exe from here and double click

**Step 2:** Click on "Run as a Local System" and then on "Next."




**Step 3:** Enter 8080 as the port number and click Next.

**Step 4:** To proceed, copy the initial password from the given path and press enter.

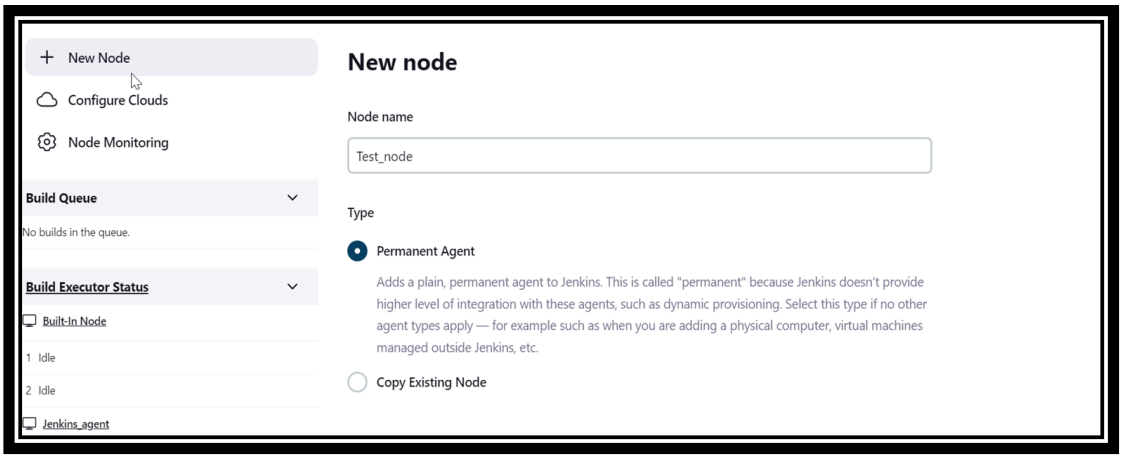



## 3.2.9.3 Connect Jenkins Node to Jenkins Master

**Step 1:** Go to "Manage Jenkins"

**Step 2:** Click on Manage nodes and clouds and enter the details like Jenkins_home path and node label as NASSCOM

**Step 3:**

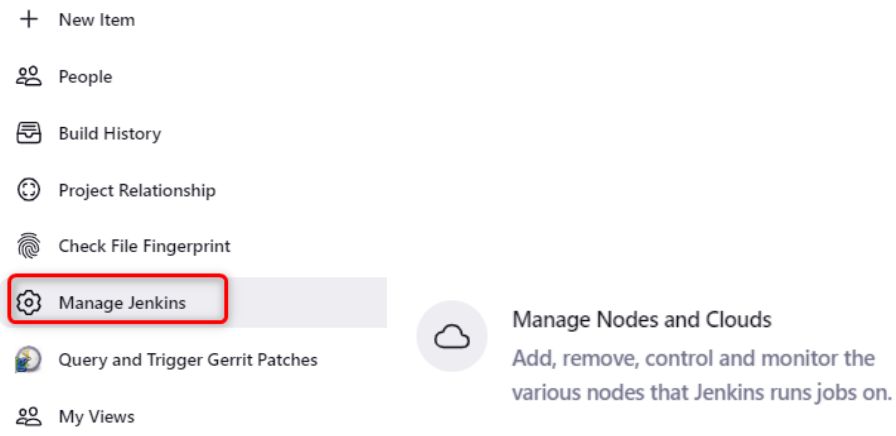


**Step 4:** Go to the Windows machine you want to connect to, access Jenkins' URL, and download agent.jar.

**Step 5:** Execute the below command


**Step 6:** Now your Jenkins node is connected to the Jenkins master.

**Step 7:** Verify your Jenkins node connection. -Go to Manage Jenkins and select Manage Nodes and Clouds, as shown in the image below.



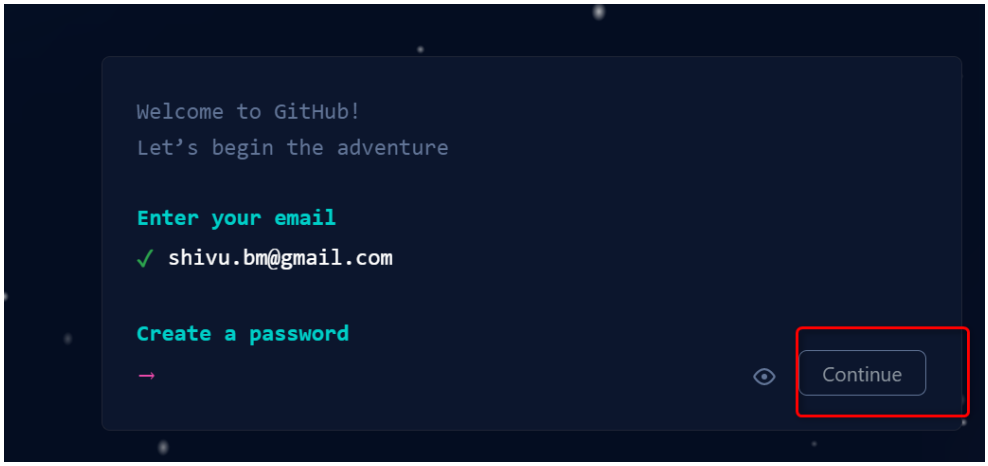

## 3.2.9.4 How to Create Github Account?

**Step 1:** Go to https://github.com/

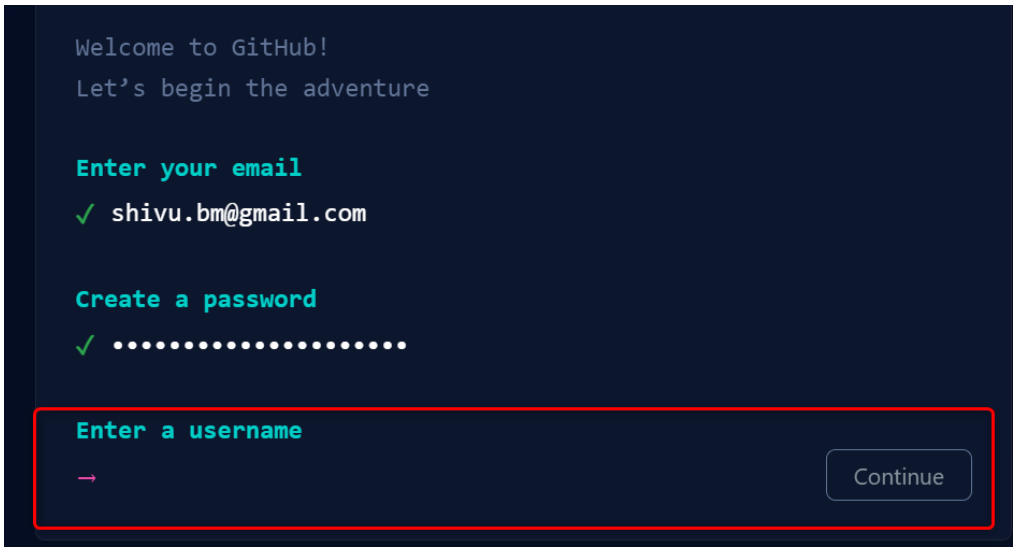
**Step 2:** In the top right corner, click the signup button.

**Step 3:** Enter your email address.

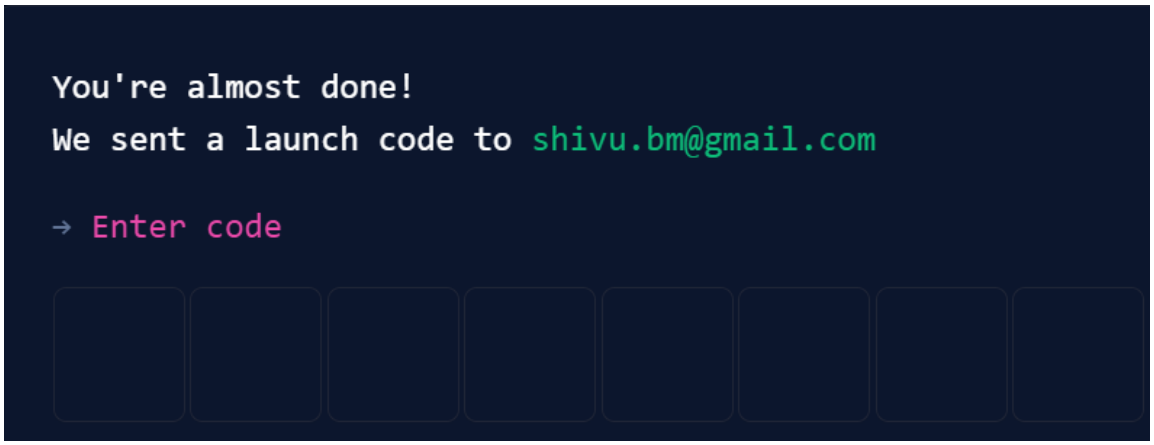
**Step 4:** Create a new password and click on "Continue"




**Step 5:** Create your GitHub username and click on "Continue."
Eg: Shvakumar-bm




**Step 6:** Check your inbox for the code you received from github.com.




**Step 7:** Now you can access the public repo
https://github.com/shivu499/DevOps-CI-CD

## 3.2.10 Jenkins Pipeline

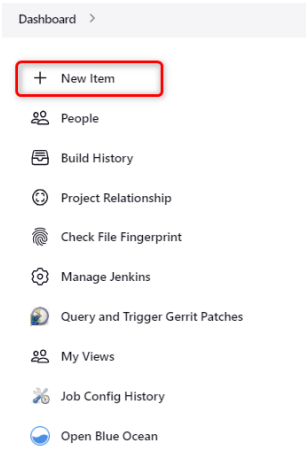
Jenkins Pipeline (or simply "Pipeline" with a capital "P") is a suite of plugins that supports implementing and integrating continuous delivery pipelines into Jenkins. A continuous delivery (CD) pipeline is an automated representation of your software delivery process, from version control to your users and customers. Every change to your software (committed in source control) goes through a complex process on its way to being released. This process involves building the software in a reliable and repeatable manner, as well as progressing the built software (called a "build") through multiple stages of testing and deployment.

## 3.2.10.1 How to Create Jenkins Pipeline Job?

**Step 1:** Go to your Jenkins

Eg: http:localhost:8080/jenkins

**Step 2:** Click on "New Item"

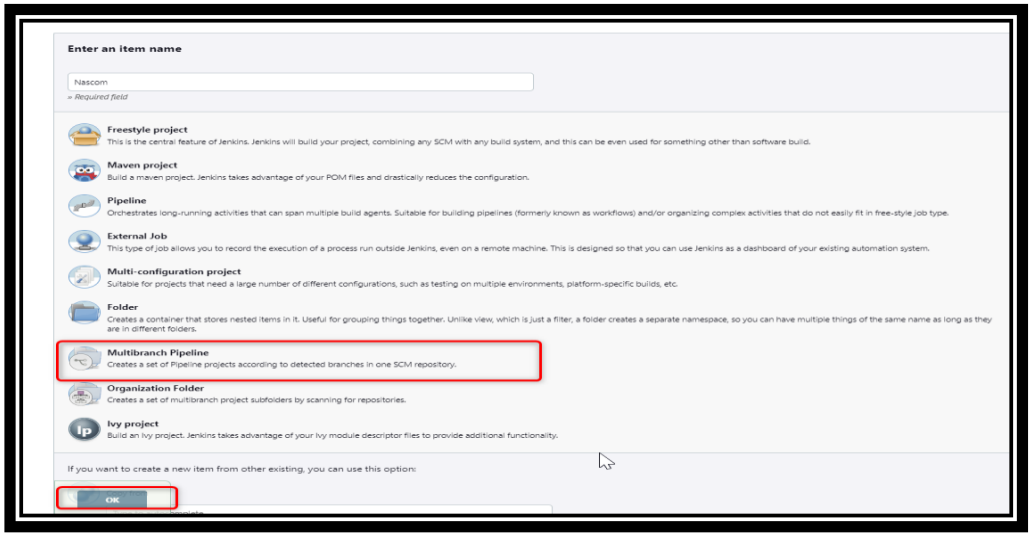



**Step 3:** Enter the job name

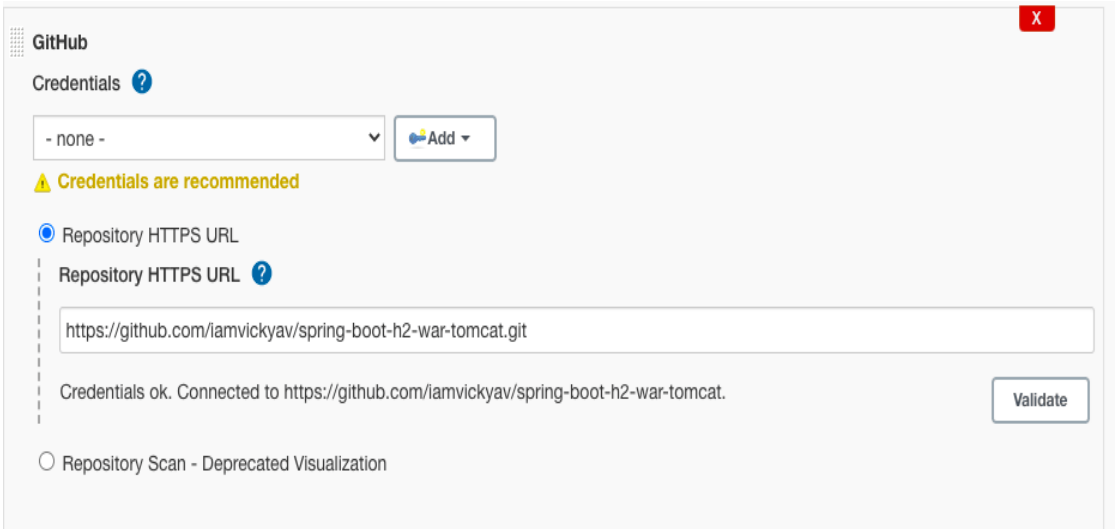
Eg: NASSCOM.

**Step 4:** Select “Multibranch Pipelines”

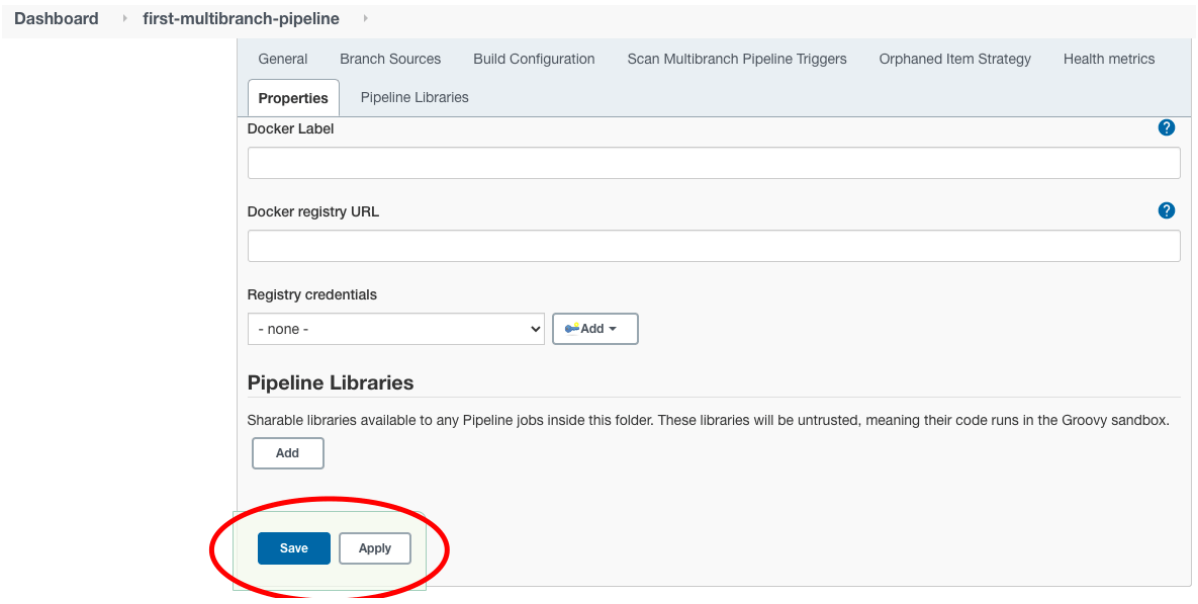
**Step 5:** Press the "OK" button.



**Step 6:** In the configure page, we need to configure only one thing: the Git repository source. Scroll down to the Branch Sources section and select the Add Source option from the dropdown.
**Step 7:** Choose GitHub as the source, as our sample GitHub repo is hosted there.
Step 8: Enter the repository HTTPS URL as https://github.com/shivu499/DevOps-CI-CD.git and click on "Validate."
Since our GitHub repository is hosted as a public repository, we don't need to configure credentials to access it. For enterprise or private repositories, we may need credentials to access them.




The "**Credentials OK**" message indicates that the connection between the Jenkins server and the Git repository is successful.
**Step 9**: Leave the rest of the configuration sections as they are for now and click on the "Save" button at the bottom.
On saving, Jenkins will perform the following steps automatically



## 3.2.10.2 Prepare Make File

- Install COMA and MINGW (download the exe file and double-click it to install it automatically).
- Set the paths in the make file, lines 1, 2, 3, and 4 in the Jenkinsfile, as shown in the image below.
- Run the pipeline and see the console output for the results.

44 lines (32 sloc) | 984 Bytes

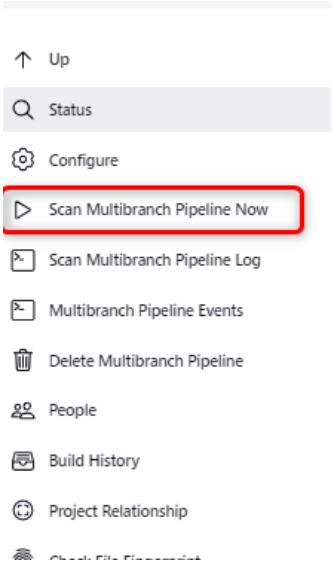
```
1  C_INCLUDE_PATH :=C:\cmocka\include
2  LD_LIBRARY_PATH :=C:\cmocka\lib
3  CC := gcc
4  GCOV = C:\Tools\MinGW\v4.8.1-1.0.0\bin\gcov
5  C_FLAGS := -Wall -Wextra -Wpedantic -std=c99 -g -O2 -fprofile-arcs -ftest-coverage
6
7  UT_TARGET :=    triangle
8  UT_SOURCE :=    ./source/triangle.c     \
9                                  ./test/test_main.c
10
11 UT_INCLUDES +=  -I$(C_INCLUDE_PATH) \
12                                 -I./source
13
14 LD_FLAGS := -L$(LD_LIBRARY_PATH)
15
16 # do not modify the following contents
17 SOURCE_NO_PATH = $(notdir $(UT_SOURCE))
18 UT_OBJ := $(SOURCE_NO_PATH:%.c=%.o)
19
20 .PHONY: all
21 all: run_test coverage
22
```

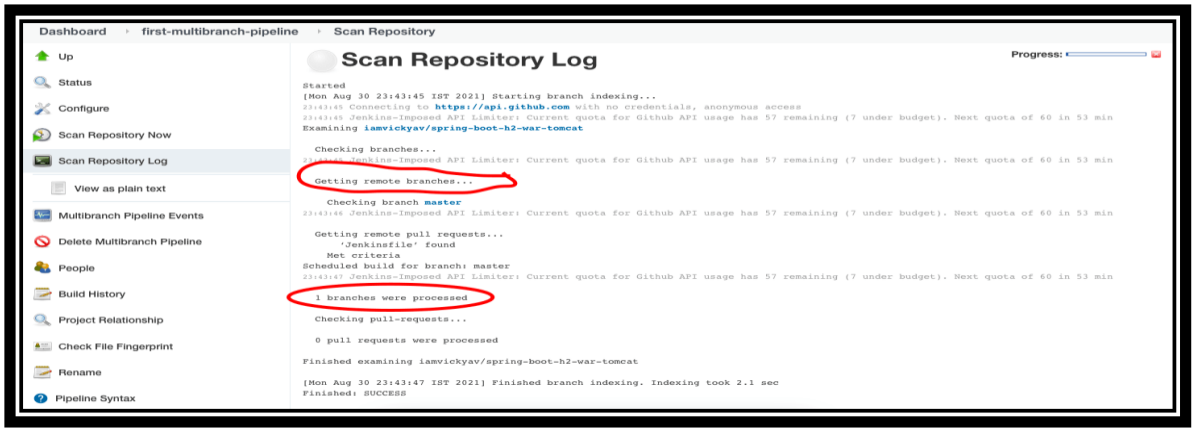
**Scan Repository Step**

- Scan the Git repository we configured.
- Look for the list of branches available in the Git repository.
- Choose branches that have a Jenkinsfile.

**Running Build Step**

- Run build for each of the branches found in previous step with steps mentioned in Jenkinsfile
- From the scan repository log section, we can understand what happened during the scan repository step.

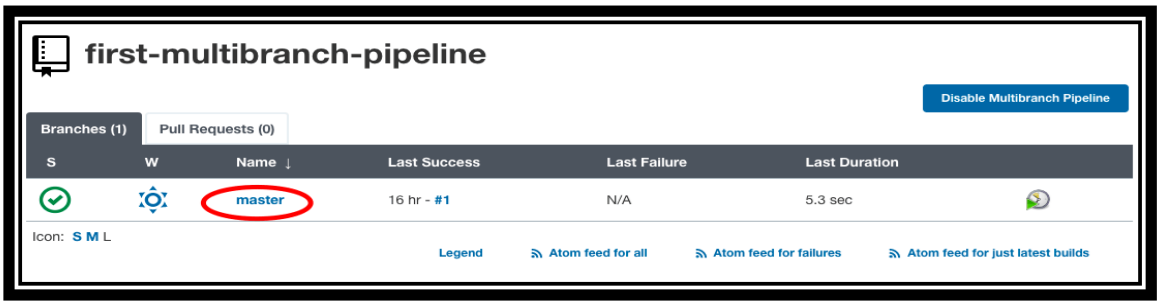




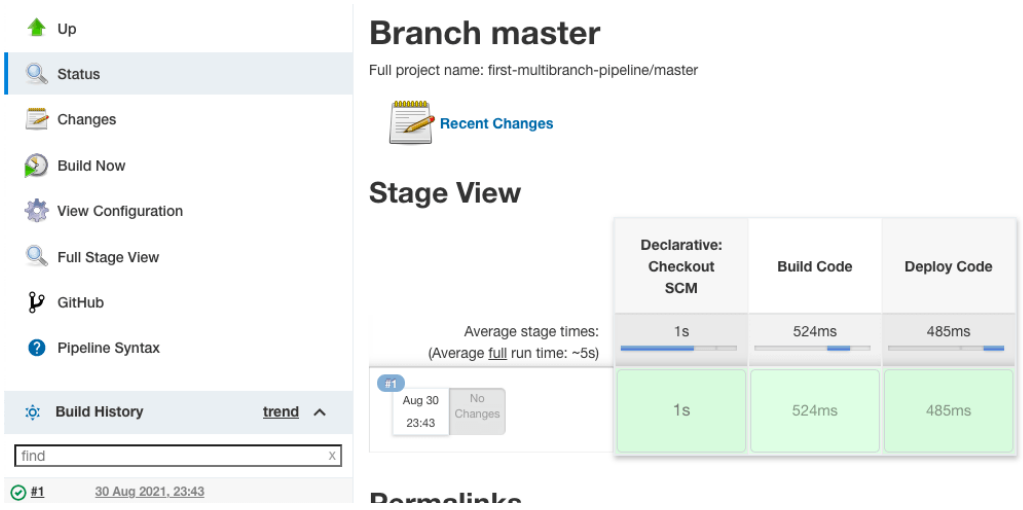

Because our git repository only contains a master branch, scan repository log reports that only one branch was processed. After the scan is complete, Jenkins will create and run a build job for each processed branch separately. In our case, we had only one branch called Master. As a result, build will only run for the master branch. We can check the same by clicking on "Status" in the left side menu.

## 3.2.10.3 How to Check Jenkins Status?

We can see a build job created for the master branch in the status section.

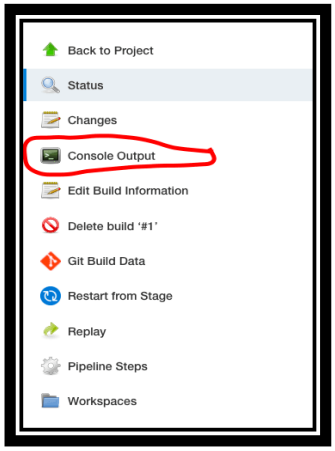



Click on the branch name to see the build job log and status.




**Stage View** gives a visual representation of how much time each stage took to execute and the status of the build job.

Choose “Console Output” from the left-side menu to see the logs.




Dashboard › first-multibranch-pipeline › master › #1

```
Checking out Revision 15540d292803ce61bfe928e7ebb6569d392afc7d (master)
 > git config core.sparsecheckout # timeout=10
 > git checkout -f 15540d292803ce61bfe928e7ebb6569d392afc7d # timeout=10
Commit message: "Update Jenkinsfile"
First time build. Skipping changelog.
[Pipeline] }
[Pipeline] // stage
[Pipeline] withEnv
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Build Code)
[Pipeline] sh
+ echo 'Building Artifact'
Building Artifact
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Deploy Code)
[Pipeline] sh
+ echo 'Deploying Code'
Deploying Code
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Jenkinsfile in develop branch:

```
pipeline {
    agent {
        node {
            // Restrict where this project can be run-testfile
            label 'nascom'
        }

        }


    stages {
        stage('Build and package') {
            steps {
                    bat """
                    make """
            }
        }
        stage('PC Lint') {
                steps {
                        echo 'PC lint execution start..'
                    bat """
                cpplint --extensions=h,c --output=junit source/triangle.c source/triangle.h 2> pc-lint.xml  """

            }

        }

            stage('Deploy To Artifactory') {

            steps{
                bat """

                  echo "upload  build release to artifactory"
                  curl -H "X-JFrog-Art-Api:AKCp8nyNzJK1TUPuvoK8vBkqd1fnkAQ5F9BZdXF49BYoeTvHk1o9FVTce1ZGCPRavQfr3bX" -T test_triangle.exe
"https://skillsync.jfrog.io/artifactory/nascom-generic-local/Releases/"

                """
            }
        }//upload full content to artifactory


    }
}
```

## 3.2.10.4 What is Pipeline?

Jenkins Pipeline (or simply "Pipeline") is a suite of plugins that supports implementing and integrating continuous delivery pipelines into Jenkins.

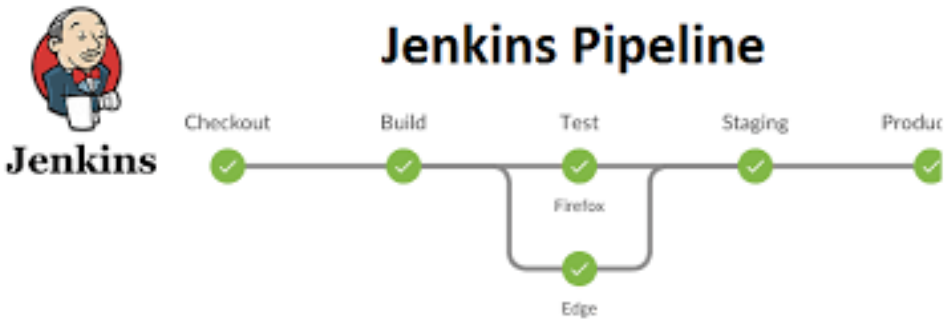



*Fig 3.2.10.4 Pipeline*

## 3.2.10.5 What is Jenkinsfile?

A Jenkinsfile is a text file that contains the definition of a Jenkins pipeline and is checked into source control. Consider the following pipeline, which implements a basic three-stage continuous delivery pipeline.

*Fig 3.2.10.5 Jenkinsfile*

The declarative pipeline example above contains the minimum necessary structure to implement a continuous delivery pipeline. The agent directive, which is required, instructs Jenkins to allocate an executor and workspace for the pipeline. Without an agent directive, not only is the declarative pipeline invalid, but it would also not be capable of doing any work! By default, the agent directive ensures that the source repository is checked out and made available for steps in the subsequent stages. The stages and steps directives are also required for a valid declarative pipeline, as they tell Jenkins what to execute and in which stage it should be executed.

## 3.2.11 Build Stage

For many projects, the beginning of "work" in the pipeline would be the "build" stage. Typically, this stage of the pipeline will be where source code is assembled, compiled, or packaged. The Jenkinsfile is not a replacement for an existing build tool such as GNU/Make, Maven, Gradle, etc. but rather can be viewed as a glue layer to bind the multiple phases of a project's development lifecycle (build, test, deploy, etc.) together. Jenkins has several plugins for invoking practically any build tool in general use, but this example will simply invoke make from a shell step (sh). The sh step assumes the system is Unix or Linux-based; for Windows-based systems, the bat step could be used instead.

```
stages {
    stage('Build and package') {
        steps {
                bat """
            make """
        }
    }
```

*Fig 3.2.11 Build Stage Sample*

## 3.2.12 Node, Stage, and Steps in Jenkinsfile

**Node:** It specifies where something shall happen. You give a name or a label, and Jenkins runs the block there.

**Stage:** It structures your script into a high-level sequence. Stages show up as columns in the pipeline stage view, with average stage times and colours for the stage result.

**Step:** It is one way to specify what shall happen. sh is of a similar quality, but it is a different kind of action. (You can also use build for things that are already specified as projects.)

## 3.2.13 Makefile

Makefile is a tool to simplify or organize code for compilation. Makefile is a set of commands (like terminal commands) with variable names and targets to create object files and remove them. In a single make file, we can create multiple targets to compile and remove binary files. Using Makefile, you can compile your project (program) an unlimited number of times.

## 3.2.14 PC-Lint

PC-Lint is a commercial software linting tool produced by Gimpel Software for the C and C++ languages. PC-Lint is a command-line tool for performing static code analysis, indicating suspicious or plain-wrong issues in source code.

- Works like a compiler
- Checks code statically ('at compile-time')
- Doesn't produce machine code
- Performs thousands of checks and will produce the appropriate warnings, so-called "issues" or "messages."

**What PC-Lint is not**

The focus is on finding C/C++ programming errors:

- No: Java, C#
- No: Metrics
- No: High-level design and architectural analysis

Inexpensive, compact, and powerful code checker:

- No: GUI/browsers/wizards
- No: Issue database
- No: License server
- No custom rules

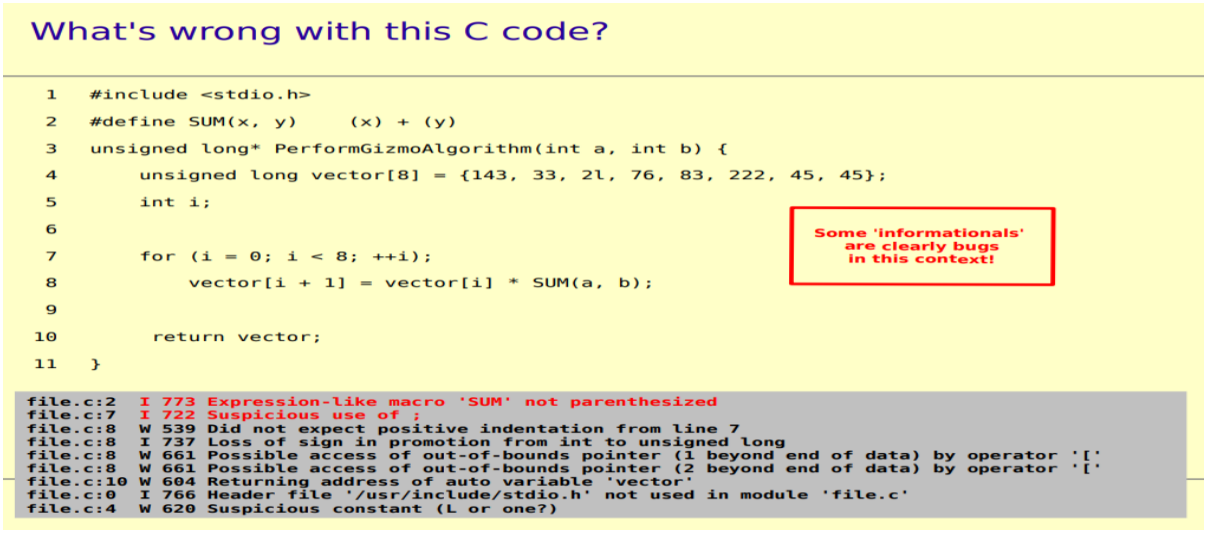



*Fig 3.2.14 PC-Lint*

## 3.2.15 MinGW

MinGW ("Minimalist GNU for Windows"), formerly mingw32, is a free and open-source software development environment to create Microsoft Windows applications. MinGW includes a port of the GNU Compiler Collection (GCC), GNU Binutils for Windows (assembler, linker, archive manager), a set of freely distributable Windows-specific header files and static import libraries that enable the use of the Windows API, a Windows native build of the GNU Project's GNU Debugger, and miscellaneous utilities. MinGW does not rely on third-party C runtime dynamic-Link library (DLL) files, and because the runtime libraries are not distributed using the GNU General Public License (GPL), it is not necessary to distribute the source code with the programs produced, unless a GPL library is used elsewhere in the program.

**Why MinGW is needed?**

MinGW is a compiler system based on the GNU GCC and Binutils projects that compiles and links code to be run on Win32 (Windows) systems. It provides C, C++, and Fortran compilers plus other related tools.

Link to download MinGW https://sourceforge.net/projects/mingw/

## 3.2.16 Unit Test

Unit testing is a type of software testing where individual units or components of software are tested. The purpose is to validate that each unit of the software code performs as expected. Unit testing is done during the development (coding phase) of an application by the developers. Unit tests isolate a section of code and verify its correctness. A unit may be an individual function, method, procedure, module, or object.

**Why perform Unit Testing?**

Unit testing is important because software developers sometimes try to save time by doing minimal unit testing, but this is a myth because inappropriate unit testing leads to high-cost defect fixing during system testing, integration testing, and even beta testing after the application is built. If proper unit testing is done in early development, then it saves time and money in the end.

Here, are the key reasons to perform unit testing in software engineering:

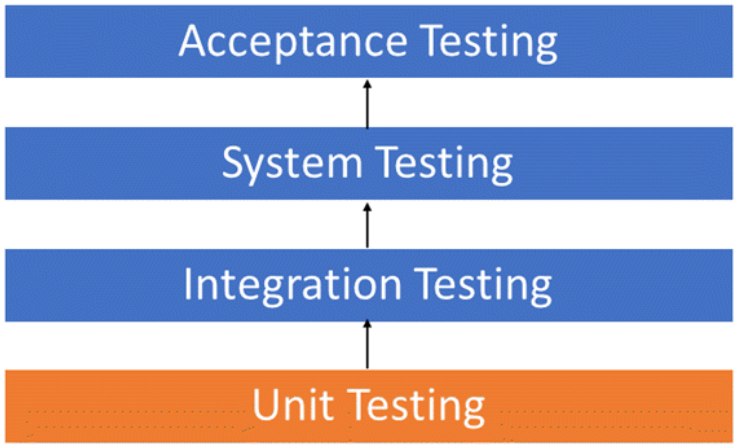



*Fig 3.2.16 Key Reasons to Perform Unit Testing*

**Advantages:**

- Unit tests help to fix bugs early in the development cycle and save costs.
- It helps the developers understand the testing code base and enables them to make changes quickly.
- Good unit tests serve as project documentation.
- Unit tests help with code reuse. Migrate both your code and your tests to your new project. Tweak the code until the tests run again.

**Best Practices of Unit Testing:**

- Unit test cases should be independent. In the event of any enhancements or changes in requirements, unit test cases should not be affected.
- Test only one code at a time.
- Follow clear and consistent naming conventions for your unit tests.
- In case of a change in code in any module, ensure there is a corresponding unit test case for the module and the module passes the tests before changing the implementation.
- Bugs identified during unit testing must be fixed before proceeding to the next phase of the SDLC.
- Adopt a "test your code" approach. The more code you write without testing, the more paths you must check for errors.

## 3.2.16.1 Unit Testing Techniques

The unit testing techniques are mainly categorized into three parts: black box testing, which involves testing the user interface along with input and output, white box testing, which involves testing the functional behavior of the software application, and gray box testing, which is used to execute test suites, test methods, test cases, and perform risk analysis. Code coverage techniques used in unit testing are listed below:

- Statement Coverage
- Decision Coverage
- Branch Coverage
- Condition Coverage
- Finite State Machine Coverage

```
1
2   #include <stdarg.h>
3   #include <stddef.h>
4   #include <setjmp.h>
5   #include <stdint.h>
6   #include <cmocka.h>
7   #include <string.h>
8   #include "triangle.h"
9
10  static int setup(void **state)
11  {
12      (void) state; /* unused */
13
14
15      return 0;
16  }
17
18  static void test_main(void **state)
19  {
20      (void) state; /* unused */
21
22      // ARRANGE
23          // uint16_t status = 0;
24
25      // ASSERT
26      assert_int_equal(TypeOfTriangle(3, 4, -8), -1);
27      // ASSERT
28      assert_int_equal(TypeOfTriangle(10, 20, 30), 0);
29      // ASSERT
30      assert_int_equal(TypeOfTriangle(-10, 20, 30), -1);
31  }
32
33  int main(void) {
34      const struct CMUnitTest tests[] = {
35          cmocka_unit_test_setup_teardown(test_main, setup, NULL)
36      };
```

*Fig 3.2.16.1 Example of Unit Test Coverage*

Here you can see that I have written a unit test case for identifying which type of triangle it is.
Line numbers 26 to 30 cover all 3 test cases of the function written in triangle.c.

## 3.2.17 Artifactory

Artifactory is a brand name to refer to a repository manager that organizes all of your binary resources. These resources can include remote artifacts, proprietary libraries, and other third-party resources. A repository manager pulls all these resources into a single location.
The word "Artifactory" refers to the JFrog product, the JFrog Artifactory, but there are several other package managers out there, such as Packagecloud.

## 3.2.18 JFrog

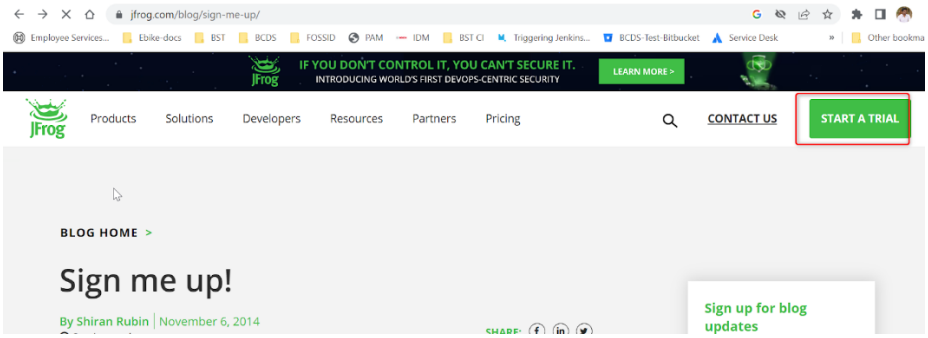
JFrog Artifactory is a universal DevOps solution that manages and automates artifacts and binaries from start to finish during the application delivery process. It gives you the option to choose from 25+ software build packages, all major CI/CD systems, and other existing DevOps tools. Artifactory is a Kubernetes and Docker registry with full CLI and REST APIs that can be customized to your ecosystem. It supports containers, Helm charts, and Docker.

## 3.2.18.1 How to Create JFrog Artifactory Account?

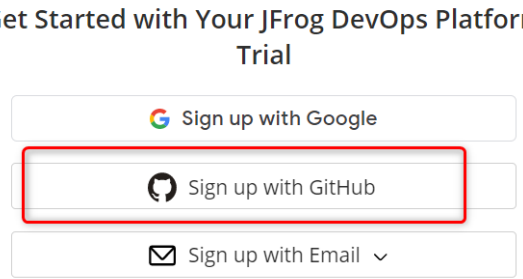
**Step 1:** Click on the below link.

https://jfrog.com/blog/sign-me-up/

**Step 2:** Click on the "Start a Free Trial" button.



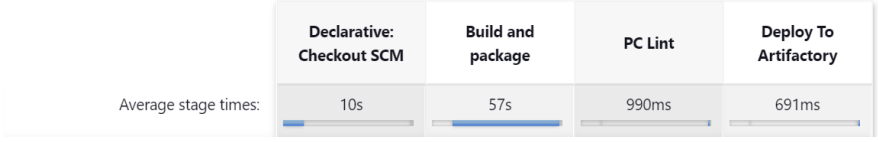

**Step 3:** Click on the "Sign up with GitHub" option and enter your previously created GitHub account details.




## 3.2.19 Pipeline Stages

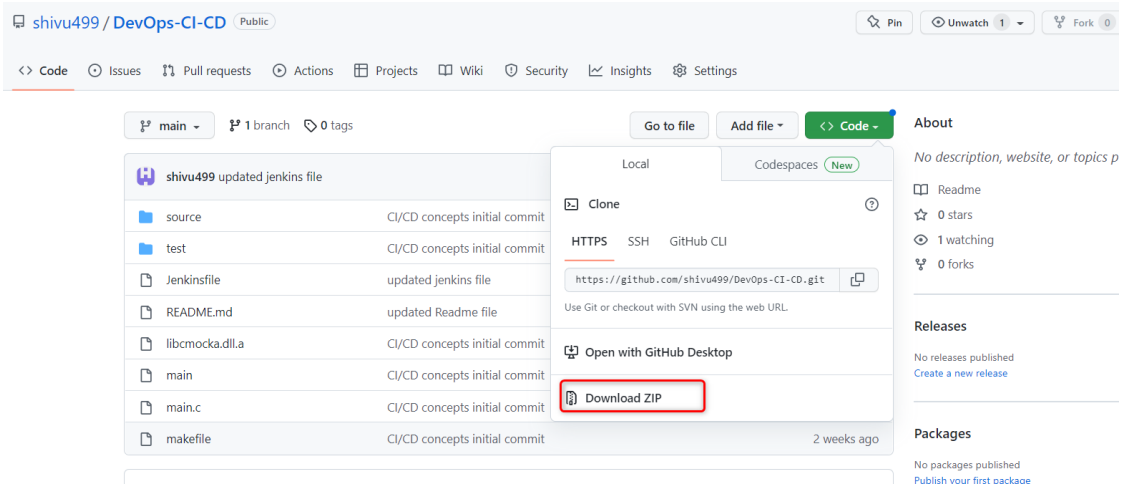
- Checkout repo
- Build and package
- Unit test
- PC Lint
- Deploy to artifactory

**Stage View**



**How to run source code manually without using jenkins?**

**Step 1:** Download the source code here: https://github.com/shivu499/DevOps-CI-CD




**Step 2:** Go to NASSCOM folder

**Step 3:** To generate, build, package, and unit test

- Make

**Step 4:** Generate PC Lint Results

- cpplint --extensions=h,c --output=junit source\triangle.c source\triangle.h 2> pc-lint.xml