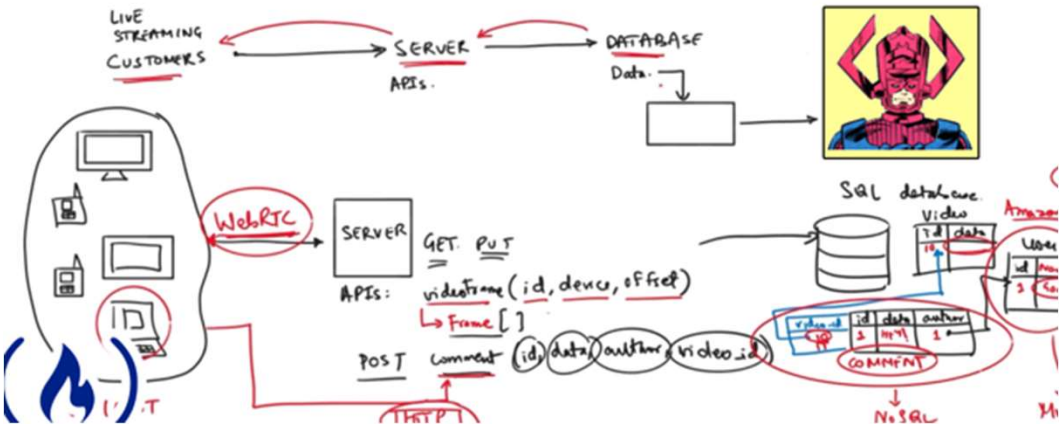
Jenkins

Lets Start the Automation Journey

By : Sandeep Kumar Sharma

What is System Design ?



In software engineering, system design is the process of defining the architecture, components, and interactions of a software system to meet specific requirements. It involves creating a blueprint that outlines how various elements work together to achieve the desired functionality, performance, and reliability

Monolith Architecture

A **monolithic architecture** is a traditional software design where all components of an application (UI, business logic, and data access) are tightly coupled and run as a single, unified codebase.

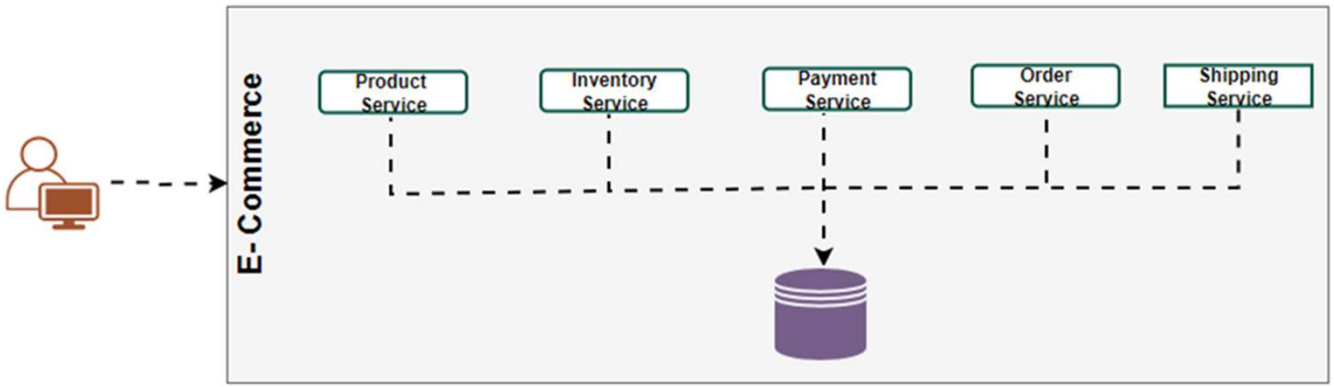
Key Characteristics:

1.Single Codebase: All functionality resides in one codebase and is deployed as a single unit.

2.Tightly Coupled Components: Changes in one part of the application often require changes in other parts.

3.Single Deployment: The entire application is built, tested, and deployed together.

4.Shared Resources: All components share a single database or resource pool.



Monolith Architecture

Advantages:

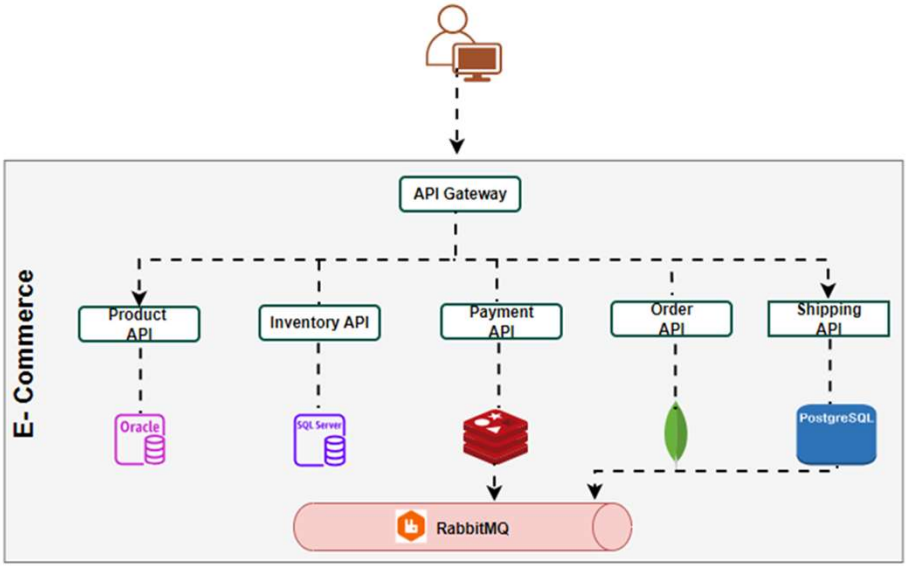
- **Simple Development:** Easy to develop, test, and deploy initially.
- **Performance:** Less overhead for inter-process communication (IPC) since everything runs in the same process.
- **Ease of Debugging:** Single application means centralized logs and stack traces.

Disadvantages:

- **Scalability Issues:** Difficult to scale specific parts of the application independently.
- **Complexity Over Time:** As the application grows, maintaining and updating the code becomes harder.
- **Slower Deployment:** A small change requires rebuilding and redeploying the entire application.
- **Lack of Flexibility:** Limited choice of technology stack since all components share the same language/framework.

Microservice Architecture

A **microservices architecture** is a modern software design pattern where an application is built as a collection of small, independent, and loosely coupled services. Each service performs a specific business function and communicates with others through APIs.



Key Characteristics:

1.Independent Services: Each service has its own codebase, database, and deployment pipeline.

2.Loosely Coupled: Changes in one service do not affect others, as they communicate through well-defined interfaces (e.g., REST, gRPC, message queues).

3.Decentralized Data Management: Each service can manage its own database, enabling different data models and technologies.

4.Technology Diversity: Teams can use different programming languages or frameworks for different services.

Microservice Architecture

Advantages:

- **Scalability:** Each service can be scaled independently based on its resource needs.
- **Flexibility:** Teams can use the best tools and technologies for each service.
- **Fault Isolation:** Issues in one service do not bring down the entire application.
- **Faster Development:** Small, autonomous teams can work on different services simultaneously.

Disadvantages:

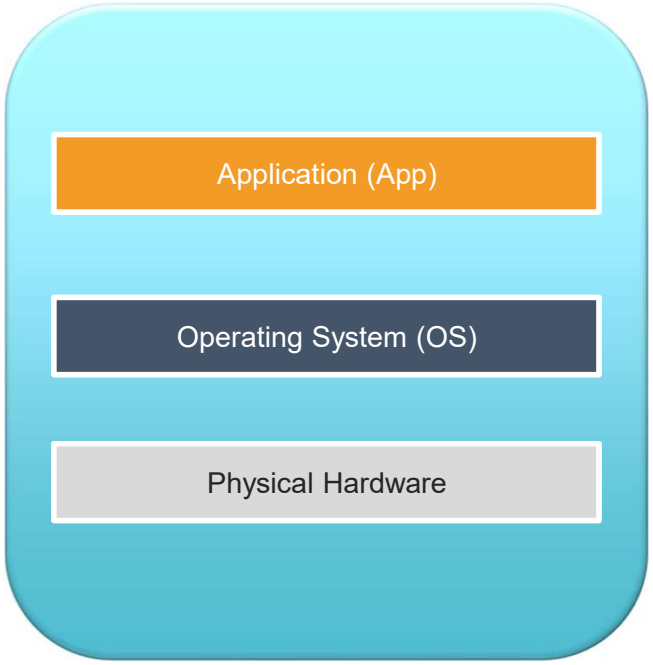
- **Complexity:** Requires robust communication mechanisms, service orchestration, and monitoring.
- **Distributed Systems Challenges:** Latency, data consistency, and service discovery become critical.
- **Higher Costs:** Multiple deployments, databases, and environments increase operational overhead.

Traditional Approach

Traditional Computing refers to the conventional approach to using physical hardware and operating systems to run applications, store data, and manage computing resources.

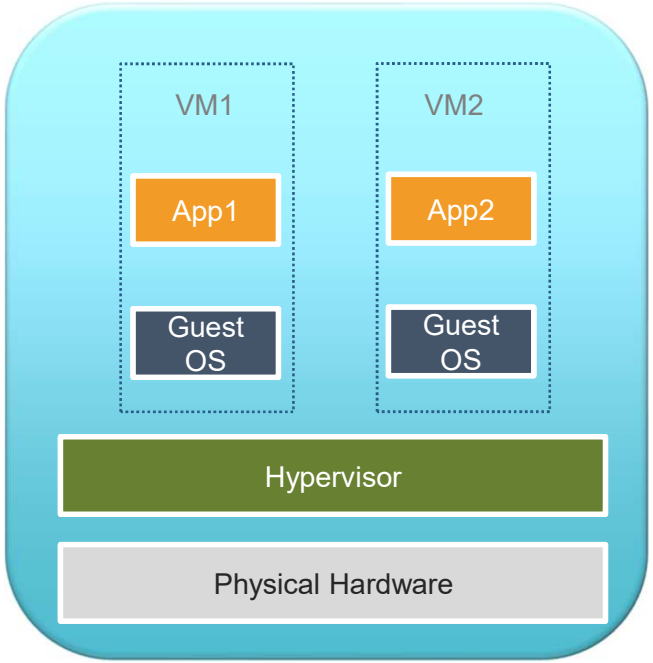
Challenges

- **Resource Underutilization**
- **Scalability Issues**
- **High Operational Costs**
- **Hardware Dependency**
- **Limited Disaster Recovery**



Traditional Approach

Virtualization



Virtualization is the process of creating a virtual version of something, typically a computing resource, such as a server, storage device, network, or operating system. Rather than relying on physical hardware alone, virtualization enables the abstraction of these physical resources to allow for greater flexibility, efficiency, and management in IT environments.

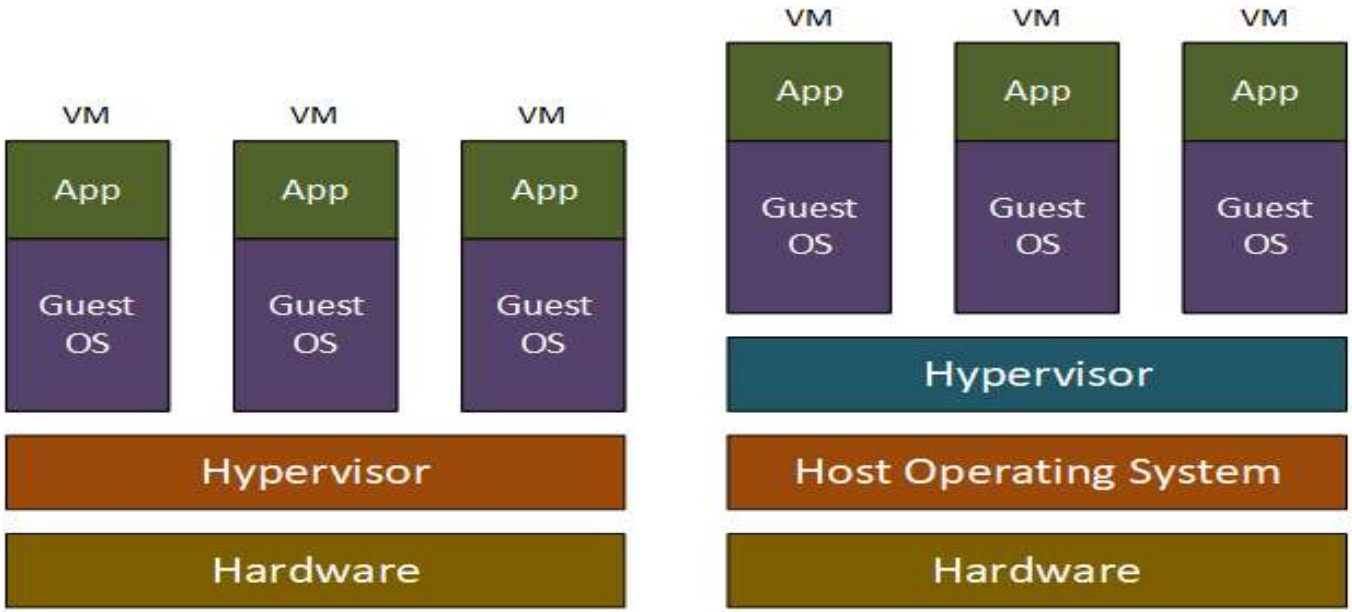
Disadvantages of Virtualization

1.Performance Overhead: Because virtualization introduces an additional software layer (the hypervisor), there may be some performance degradation compared to running applications directly on physical hardware.

2.Single Point of Failure: In environments where multiple VMs rely on a single physical server, if that server fails, all VMs can go down.

3.Security: Virtual machines share the same physical infrastructure, so vulnerabilities in the hypervisor could potentially lead to attacks across multiple virtual environments.

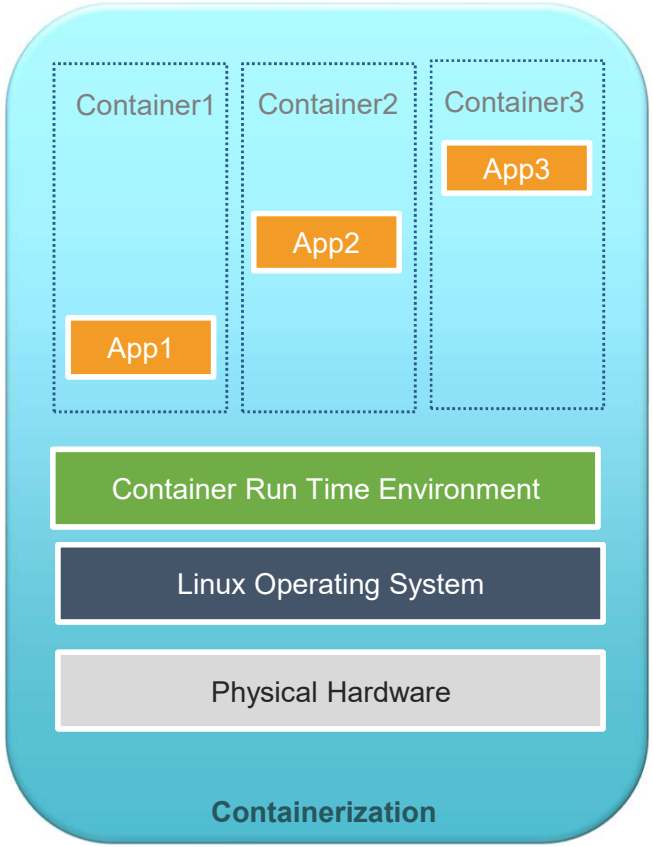
Virtualization



Type 1 Hypervisor

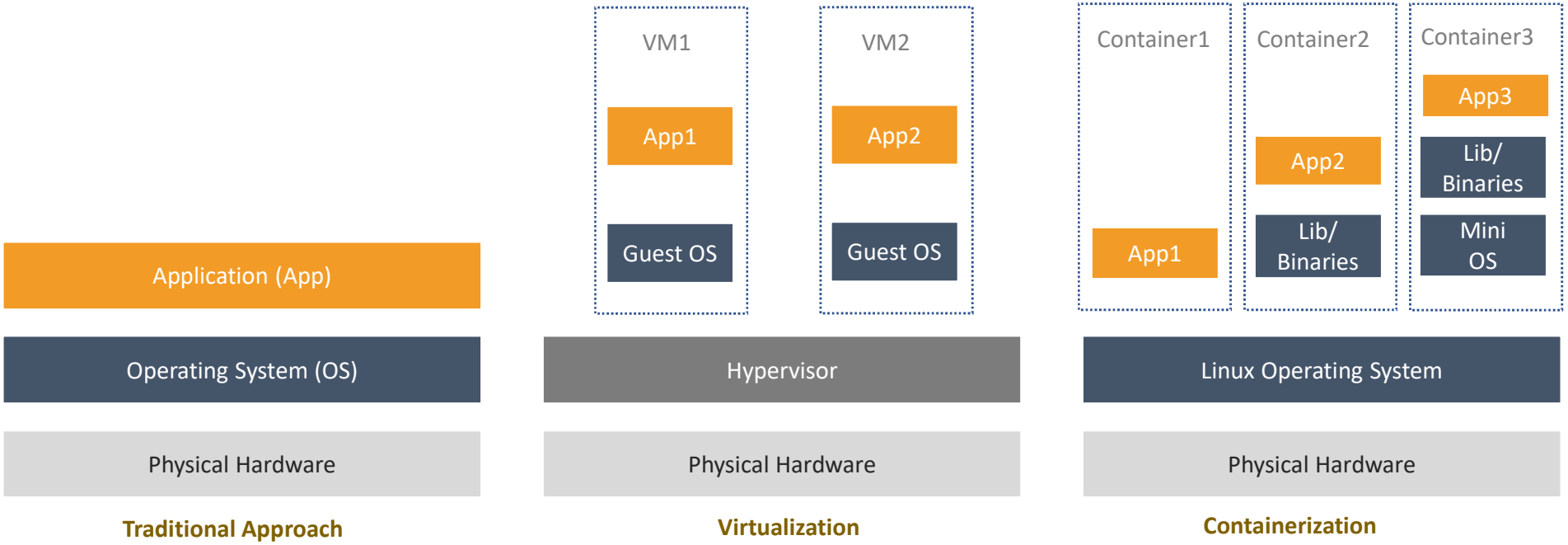
Type 2 Hypervisor

Containerization



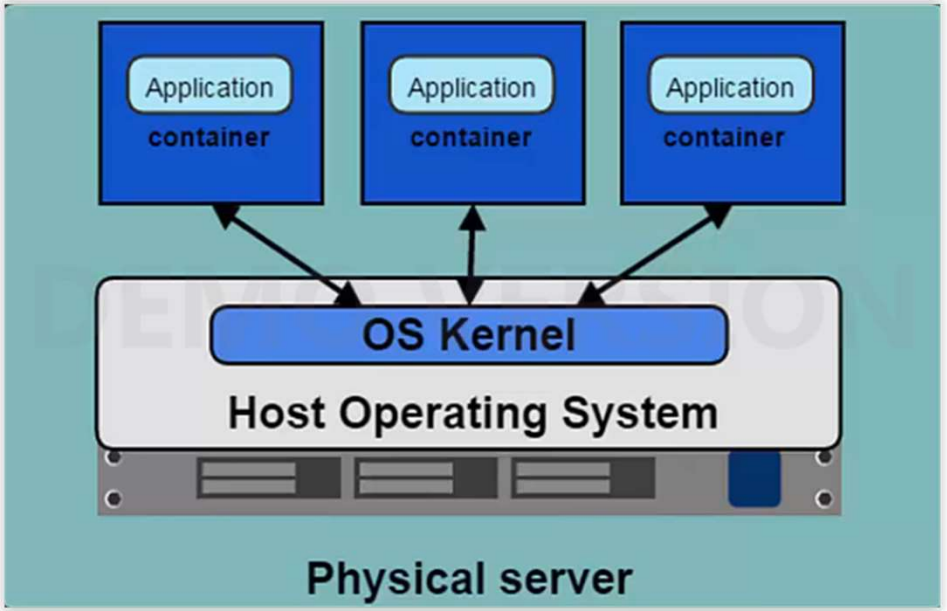
Containerization is a form of virtualization that packages applications and their dependencies into isolated environments called containers. Unlike traditional virtualization, which involves running multiple operating systems on a single host, containerization allows multiple containers to run on a single operating system instance. Each container shares the host operating system kernel but operates in its own user space, ensuring applications are isolated from each other.

Containers


Application (App)
Operating System (OS)
Physical Hardware
Traditional Approach
VM1
App1
Guest OS
VM2
App2
Guest OS
Hypervisor
Physical Hardware
Virtualization
Container1
App1
Container2
App2
Lib/ Binaries
Container3
App3
Lib/ Binaries
Mini OS
Linux Operating System
Physical Hardware
Containerization

Introducing Containers

- Container based virtualization uses the kernel on the host's operating system to run multiple guest instances
- Each guest instance is called a "Container"
  - Each container has its own
  - Root Filesystem
  - Processes
  - Memory
  - Devices
  - Network Ports
- From outside its looks like a VM but its not a VM



Introducing Containers

A container is a lightweight, portable, and executable software package that includes everything needed to run a piece of software, including the code, runtime, system tools, libraries, and settings. Containers provide a consistent and isolated environment for applications, ensuring that they run reliably and consistently across various computing environments.

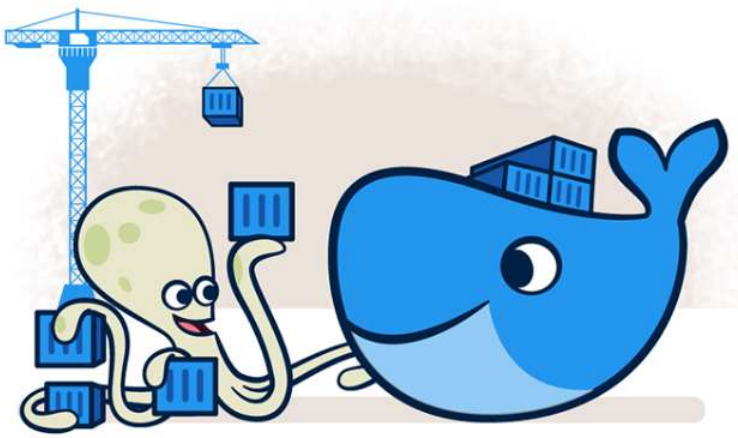
- Container are more light weight
- No need to install dedicated guest OS,
- Stop/Start time is very fast
- Less CPU, RAM, Storage Space required
- More containers per machine than VM's
- Great Portability

Key characteristics of containers

1. **Isolation:** Containers encapsulate an application and its dependencies, ensuring that they run in an isolated environment. This isolation helps prevent conflicts between different applications or different versions of the same application.
2. **Portability:** Containers can run consistently across different environments, such as development, testing, and production. This portability is possible because containers include all the dependencies needed for an application to run, reducing compatibility issues.
3. **Efficiency:** Containers share the host operating system's kernel and resources, making them lightweight compared to virtual machines. They start up quickly and consume fewer resources, allowing for more efficient use of hardware.
4. **Scalability:** Containers are well-suited for scalable and distributed applications. They can be easily deployed and managed using container orchestration tools like Kubernetes, which automates the deployment, scaling, and management of containerized applications.
5. **Versioning and Reproducibility:** Containers can be versioned, allowing developers to package and distribute applications with specific dependencies and configurations. This ensures that the software behaves consistently across different environments.

Docker

- Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly.
- With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code, you can significantly reduce the delay between writing code and running it in production.



The Docker Platform

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security lets you run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so you don't need to rely on what's installed on the host. You can share containers while you work, and be sure that everyone you share with gets the same container that works in the same way.

Docker provides tooling and a platform to manage the lifecycle of your containers:

- ✓ Develop your application and its supporting components using containers.
- ✓ The container becomes the unit for distributing and testing your application.
- ✓ When you're ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

Docker is a platform to Build, Ship and Run containerized applications

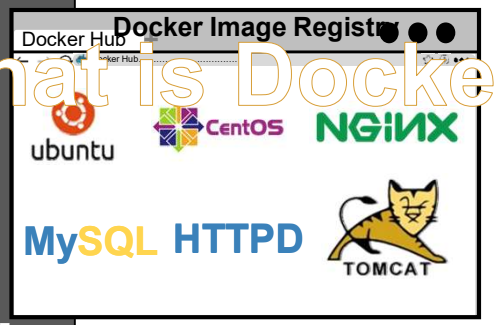
```
Dockerfile

FROM    centos:latest
RUN     yum -y update
RUN     yum -y install httpd
RUN     echo "Hello Docker" >
/var/www/html/index.html
EXPOSE 80
CMD     ["httpd", "-D", "FOREGROUND"]
```

BUILD

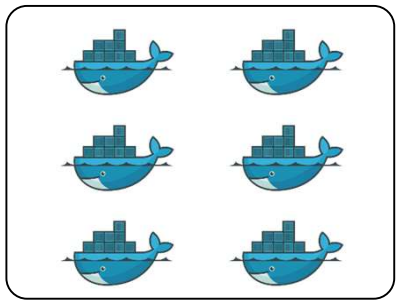
Docker File

SHIP



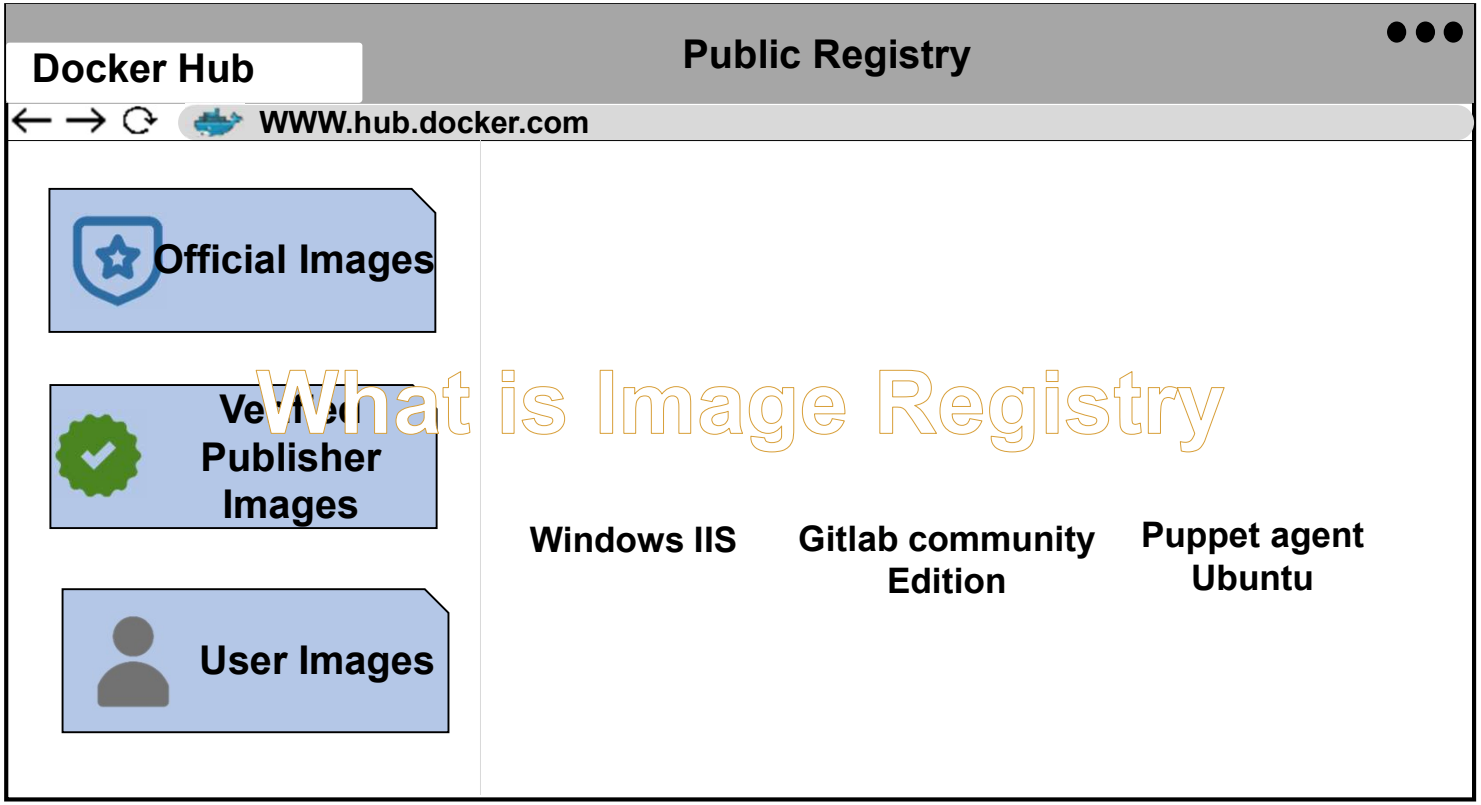
Docker Images

RUN



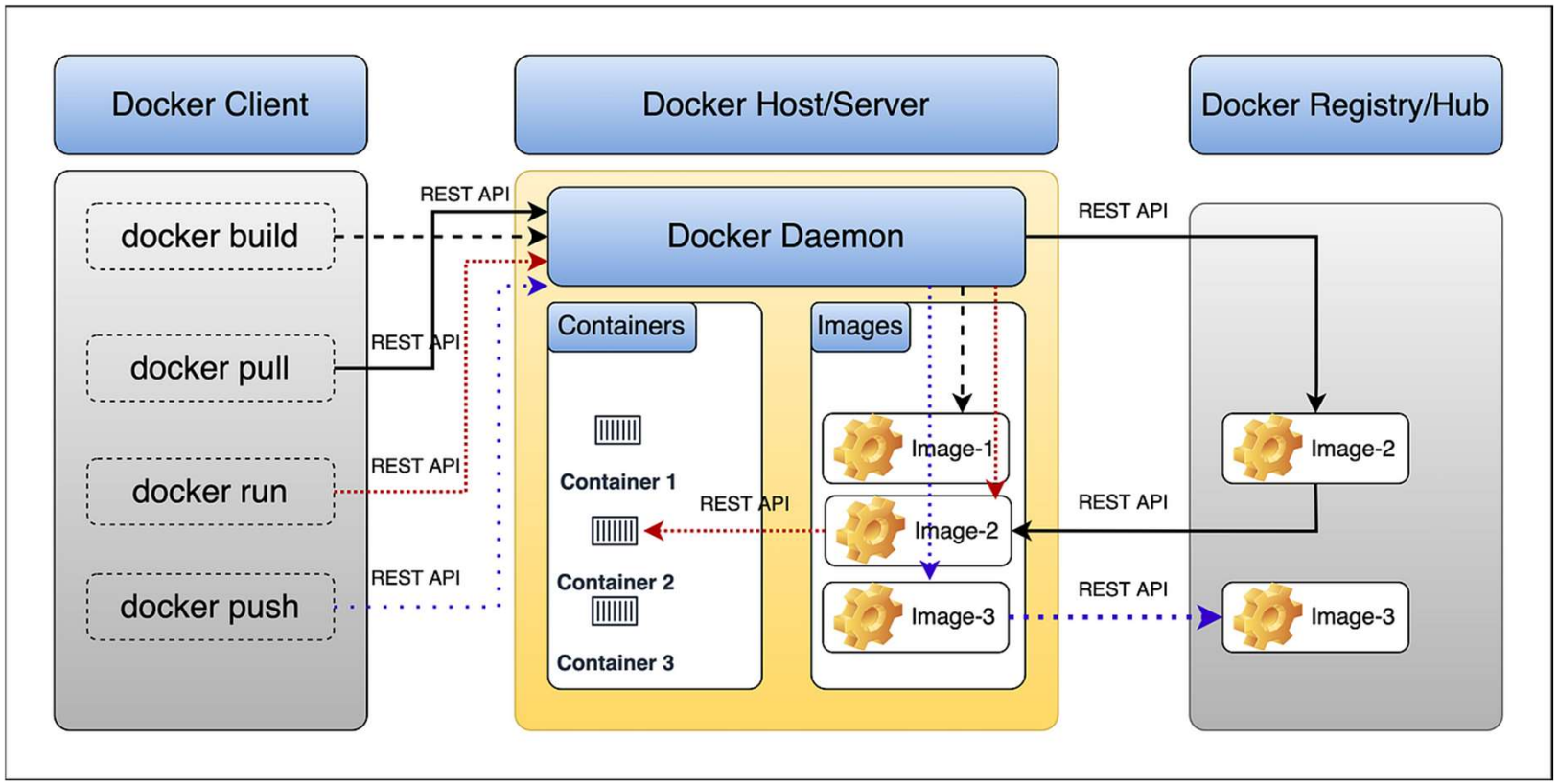
Docker Containers

What is Docker?

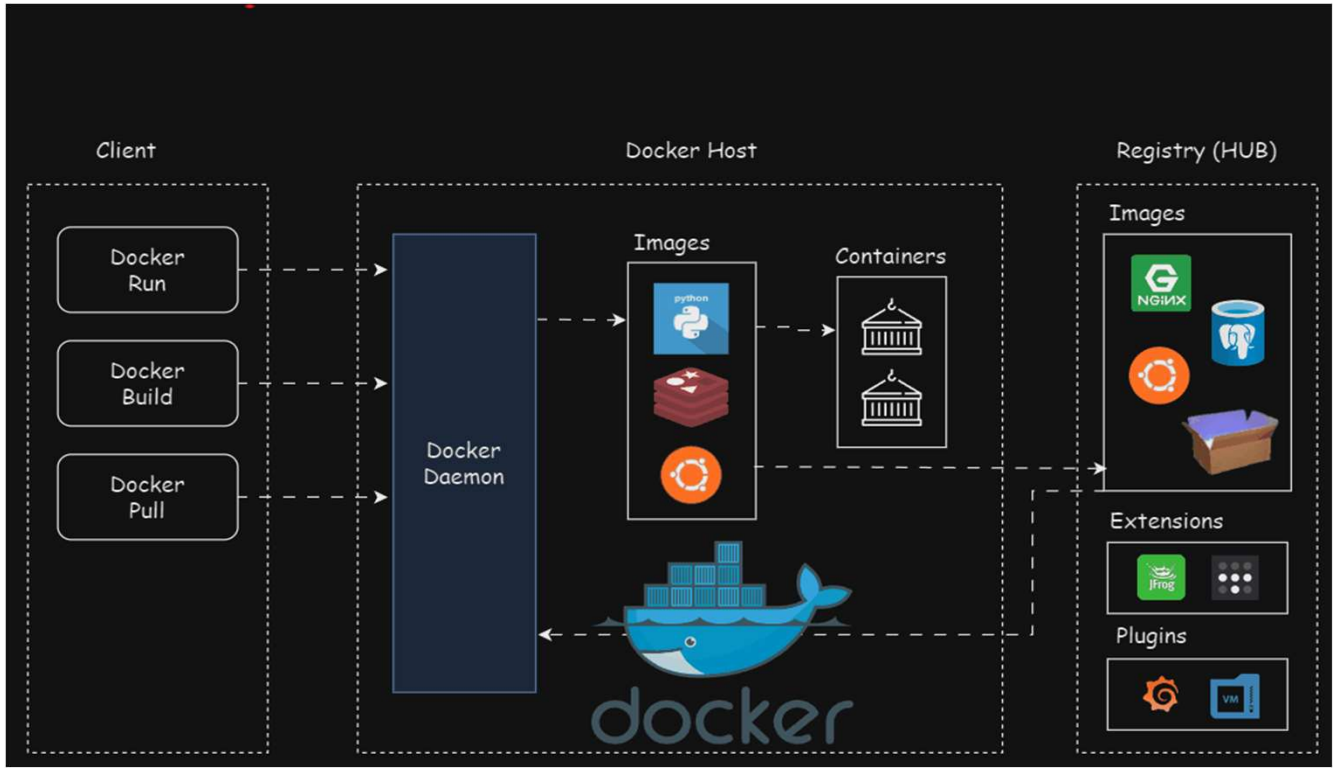

Docker Hub
Public Registry
WWW.hub.docker.com
Official Images
Publisher Images
User Images
What is Image Registry
Windows IIS
Gitlab community Edition
Puppet agent Ubuntu

Docker Architecture

Docker Architecture



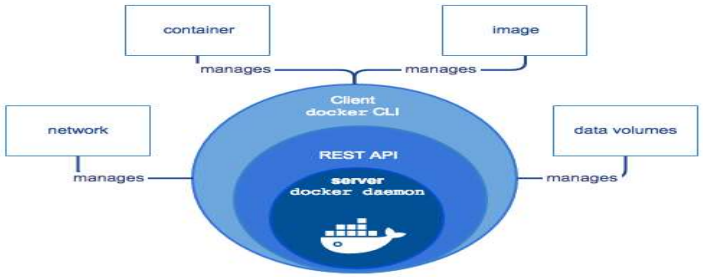
Docker Architecture

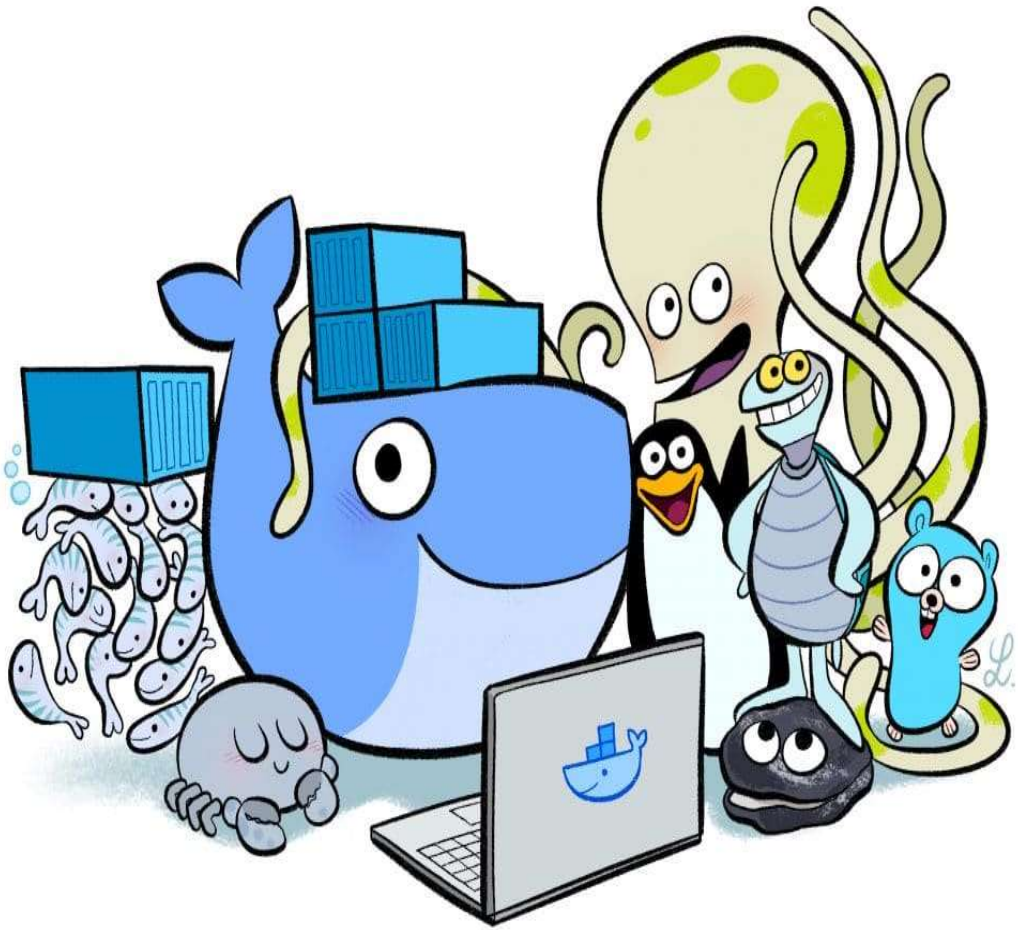


Docker Engine

The Docker Engine is the core component responsible for creating and managing containers. It includes several subcomponents, such as the Docker daemon, REST API, and the Docker command-line interface.

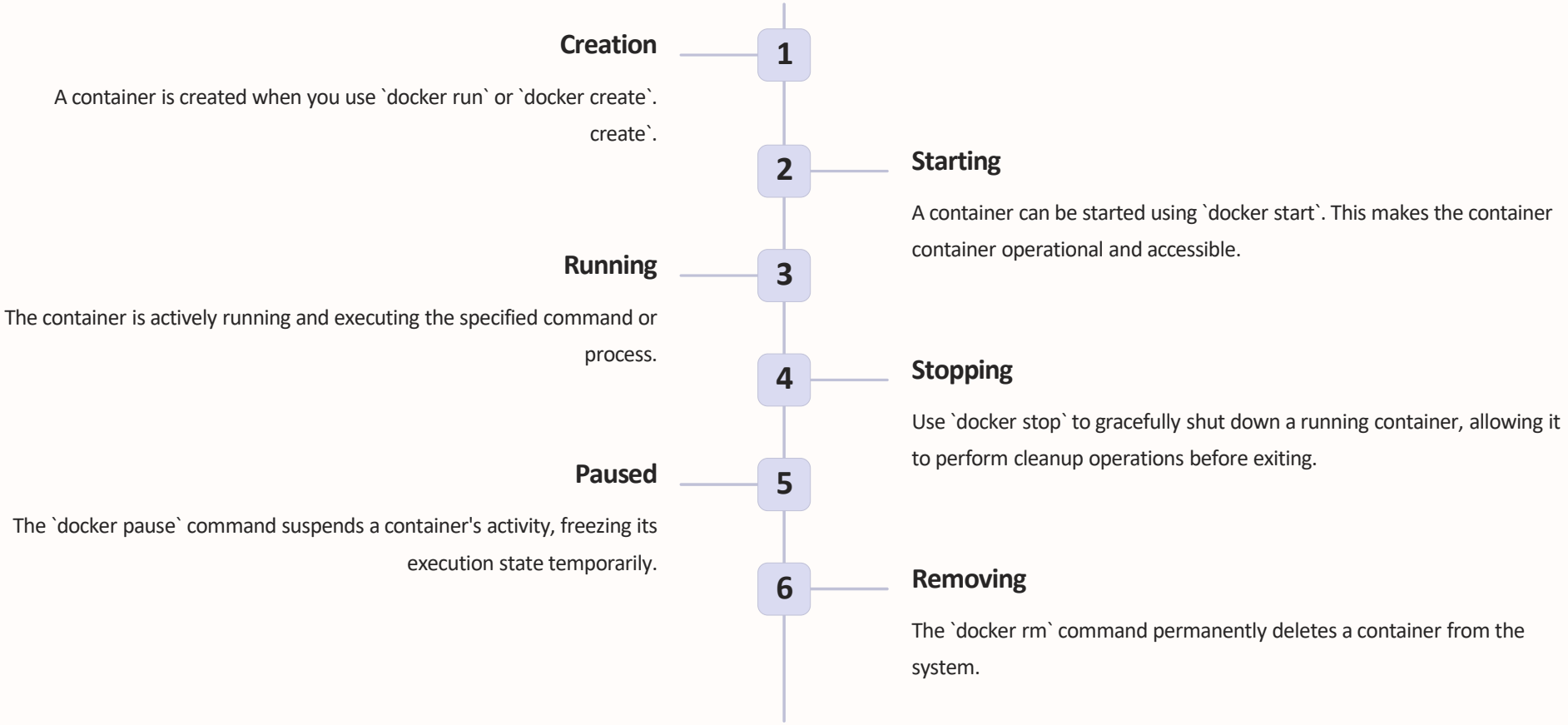
- Docker Daemon: This is a background process that manages Docker containers on a host system. It listens for Docker API requests and manages container lifecycles.
- Docker API: The API allows interaction with the Docker daemon, enabling users and applications to control Docker containers programmatically..





How to Create Containers

Docker Container Lifecycle Management



Docker Pull Command

Download Images

The `docker pull` command downloads images from a registry, making them available for use on your local system.

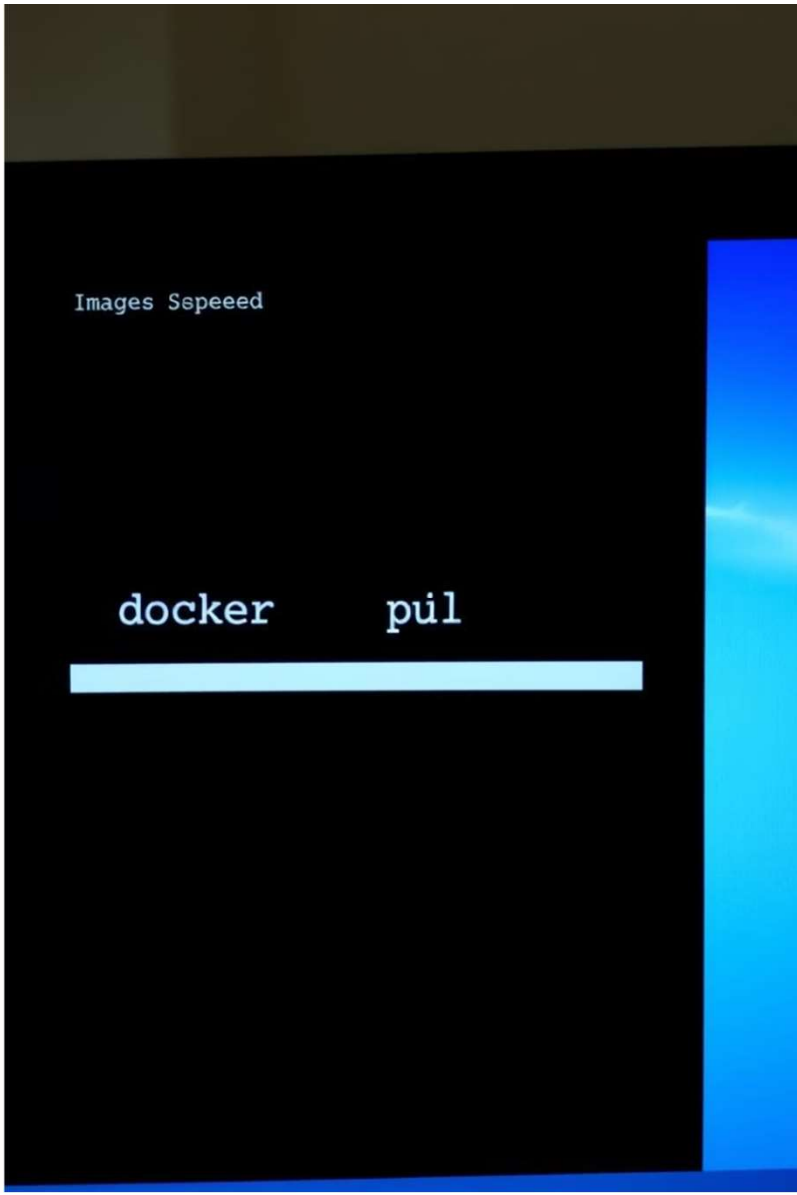
Image Naming

Images are typically named using a `/` format. The repository identifies the source, and the tag specifies a specific version.

Example

To download the latest version of the Nginx image, use:

```
docker pull nginx:latest
```



Docker Run and Create Commands

Docker Run

The `docker run` command creates and starts a Docker container. It accepts a wide array of flags for configuring the container's resources and behavior.

- `-d`: Detached mode, run in the background
- `-it`: Interactive mode, attach to the container's terminal
- `-p`: Port mapping, expose container ports
- `-v`: Volume mounting, share data between host and container

Docker Create

Creates a new container but does not start it immediately.
It prepares a container from an image but leaves it in a stopped state.
You will need to start the container manually using the docker start command.

Practical Examples

```
docker run ubuntu echo "Hello, World!"
```

This command will:

- Create a new container using the ubuntu image.
- Start the container.
- Run the command echo "Hello, World!".

```
docker create ubuntu
```

This command will:

- Create a container from the ubuntu image.
- The container will not start automatically

Docker ps Commands

Docker ps

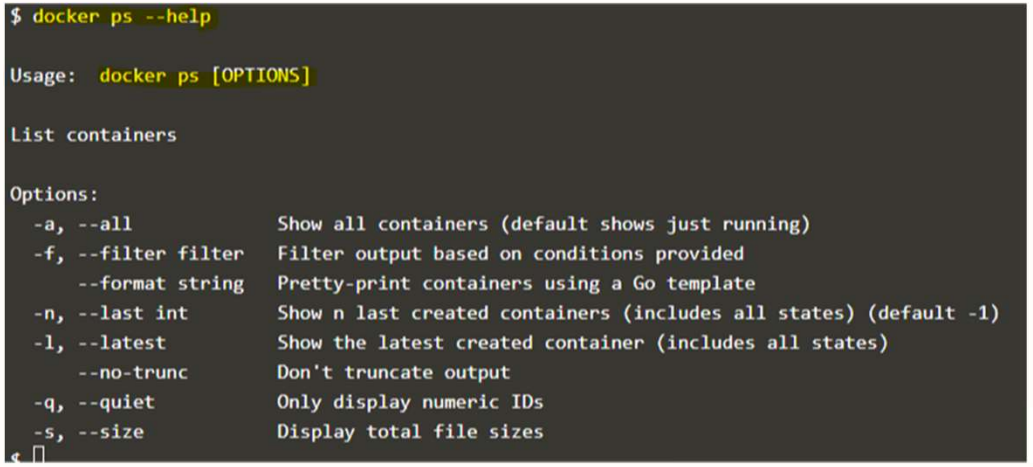
The `docker ps` command lists all running Docker containers. It also allows you to filter and sort containers based on various criteria.

- `-a`: Show all containers, including stopped ones
- `-l`: Show the most recently created container
- `-n`: Show the last N containers
- `-f`: Filter containers based on various criteria

Practical Examples

To list all containers, including stopped ones, use:

```
docker ps -a
```



Create Container -Lab

Docker Attach and Exec Commands

1

Docker Attach

The `docker attach` command attaches to a running container's standard input, output, and error streams.

2

Docker Exec

The `docker exec` command executes a command inside a running container.

3

Use Cases

Use `docker attach` to interact with the container's terminal in real-time. Use time. Use `docker exec` to run commands within the container's environment.

Docker rm and rmi Commands

1 **Docker rm**

The `docker rm` command removes one or more Docker containers. It's used to clean up unused containers and free up resources.

2 **Docker rmi**

The `docker rmi` command removes one or more Docker images. It's used to delete images that are no longer needed.

3 **Force Removal**

You can use the `-f` flag to force the removal of containers or images, even if they are running or have dependent containers.

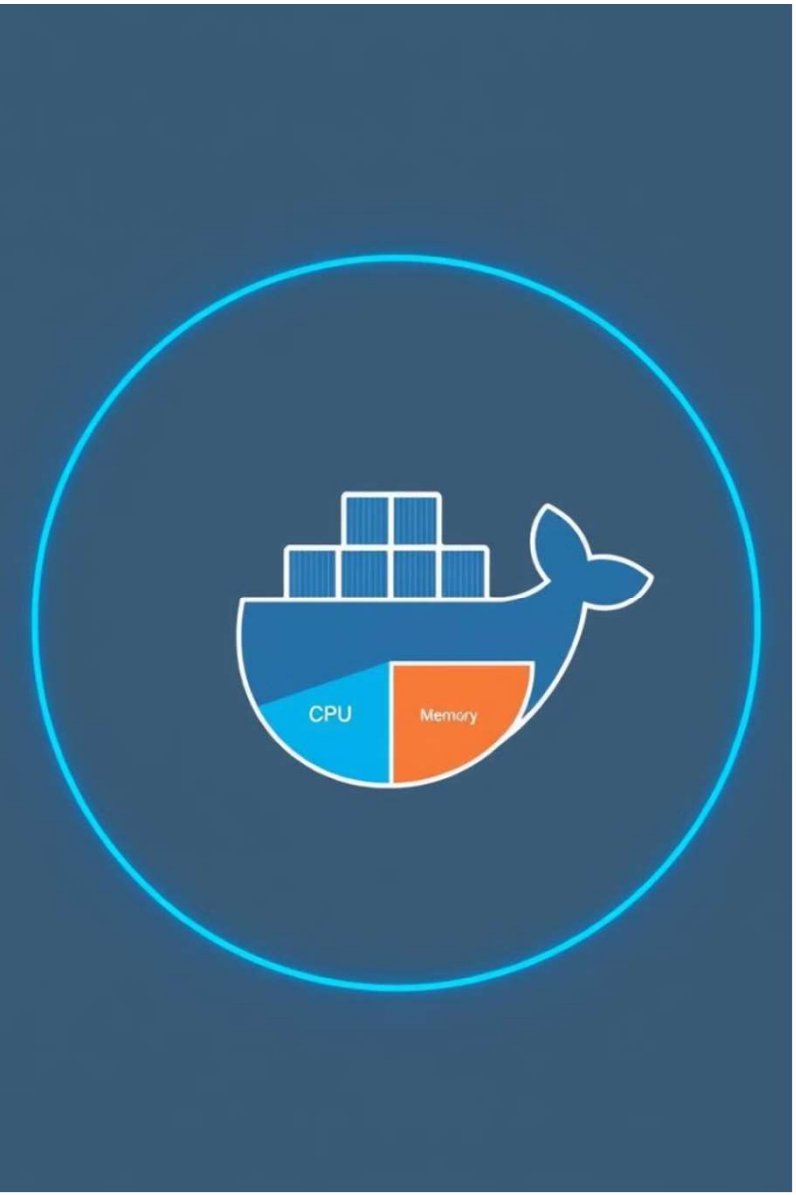
4 **Example Usage**

To remove a container named `my-container`, use:

```
docker rm my-container
```

To remove an image with the tag `nginx:latest`, use:

```
docker rmi nginx:latest
```



Docker Container Resource Management

Resource	Flag	Description
CPU	`--cpus`	Set the number of CPU cores allocated to the container.
Memory	`--memory`	Specify the maximum amount of memory the container can use.
Storage	`--storage-opt`	Configure storage options for the container, such as size limits or storage drivers.

Docker Images

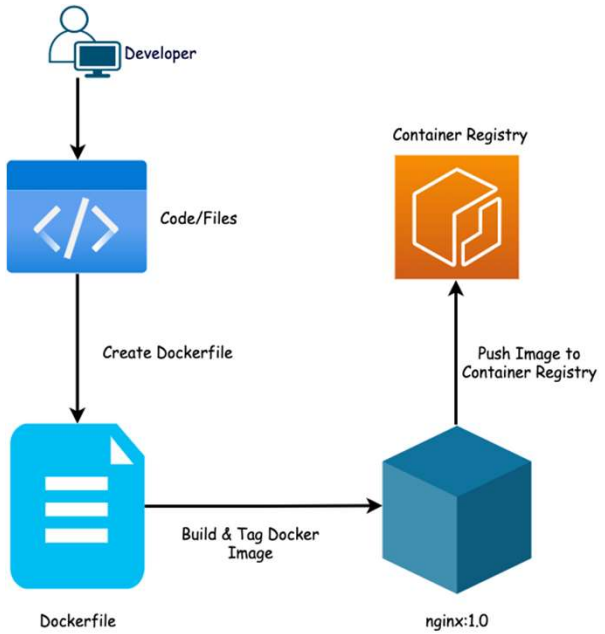
Docker Images

Docker images are lightweight, standalone, and executable packages that contain everything needed to run an application, including the code, runtime, libraries, and system tools.

We can create docker images by two way.

1. Using Running Containers

2. Using Docker file



The Process of Creating Images From Containers

At a high level, the process of creating a Docker image from a container involves three steps:

1.**Create a container from an existing image:** The first step is to choose a base image you want to customize and run it as a container.

2.**Make changes to the container:** Once you have the container up and running, you make changes to it. You could modify files, install additional software or do whatever you need to meet your requirements.

3.**Commit the changes to create a new image:** After you've made the desired changes to the container, the next step is to commit those changes to create a brand-new Docker image. Now, you can use the new image to spin up new containers with your customizations.

command

docker container commit -a "sandeep" -m "Changed default c1 welcome message" c1 image1

Image Creation Using Container Lab

Docker File

- Docker builds images automatically by reading the instructions from a Dockerfile which is a text file that contains all commands, in order, needed to build a given image.
- A Docker file is a text document that contains all the commands a user could call on the command line to assemble an image.
- Docker containers are lightweight, portable, and consistent environments that encapsulate an application and its dependencies.
- The default filename to use for a Dockerfile is Dockerfile, without a file extension.
- Using the default name allows you to run the docker build command without having to specify additional command flags

Docker File Syntax

```
# syntax=docker/dockerfile:1


FROM ubuntu:22.04

COPY . /app

RUN make /app

CMD python /app/app.py
```

Create Image Using Docker File

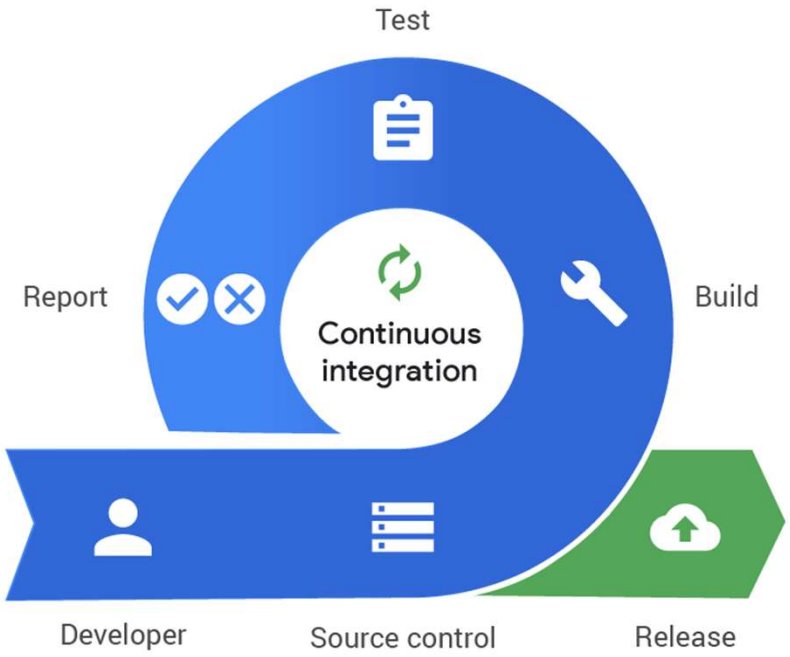
Step 1: Create a Dockerfile: Write a Dockerfile that defines the instructions for building your image. Place this file in the root directory of your project.

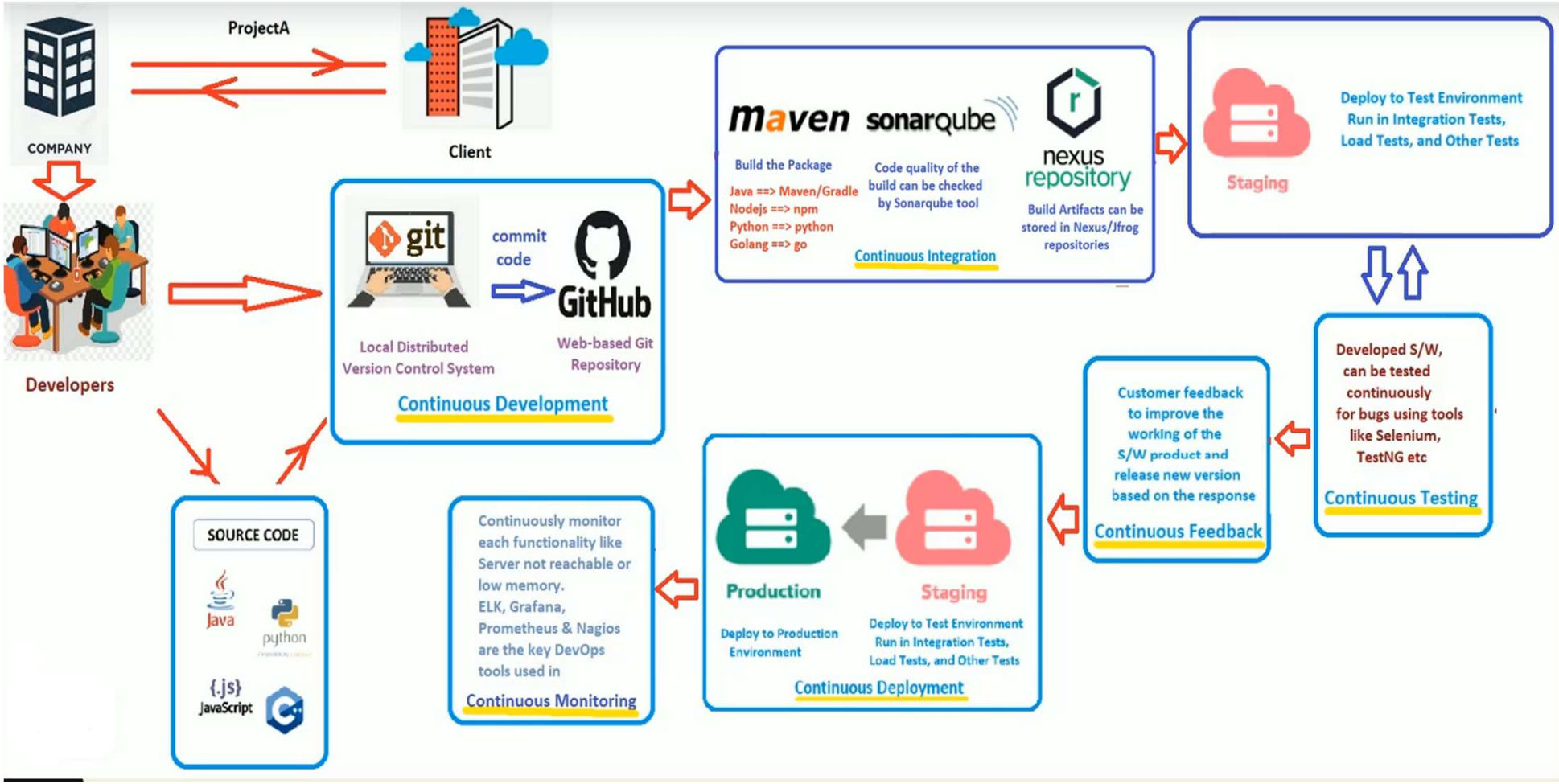
Step 2: Build the Docker Image: Open a terminal, navigate to the directory containing your Dockerfile, and run the following command to build the Docker image.

docker build -t yourimage.

Continuous Integration

Continuous Integration




ProjectA
COMPANY
Client
Developers
git
commit code
GitHub
Local Distributed Version Control System
Web-based Git Repository
Continuous Development
maven
sonarqube
nexus repository
Build the Package
Java ==> Maven/Gradle
Nodejs ==> npm
Python ==> python
Golang ==> go
Code quality of the build can be checked by Sonarqube tool
Build Artifacts can be stored in Nexus/Jfrog repositories
Continuous Integration
Staging
Deploy to Test Environment
Run in Integration Tests,
Load Tests, and Other Tests
Developed S/W, can be tested continuously for bugs using tools like Selenium, TestNG etc
Continuous Testing
Customer feedback to improve the working of the S/W product and release new version based on the response
Continuous Feedback
Production
Staging
Deploy to Production Environment
Deploy to Test Environment
Run in Integration Tests,
Load Tests, and Other Tests
Continuous Deployment
Continuously monitor each functionality like Server not reachable or low memory.
ELK, Grafana, Prometheus & Nagios are the key DevOps tools used in
Continuous Monitoring
SOURCE CODE
Java
python
{.js}
JavaScript

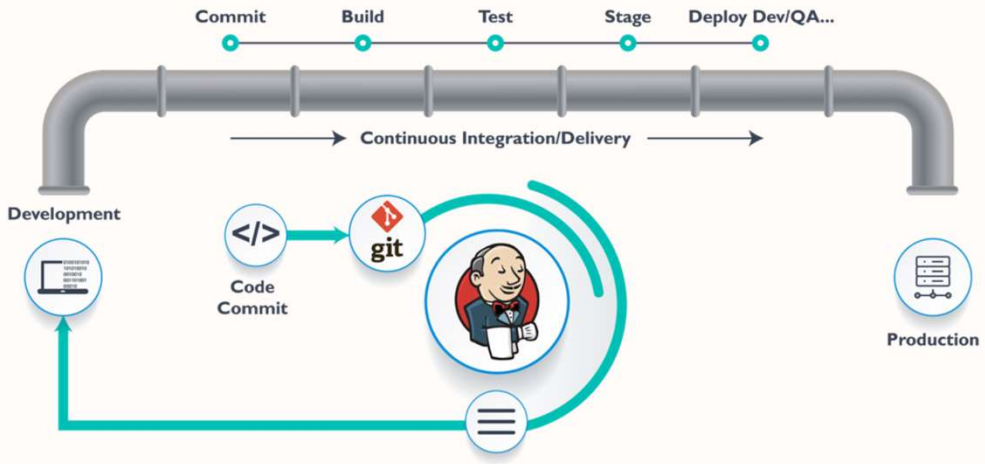
What is CI?

- 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 (CI) is a development practice in which the developers are needs to commit changes to the source code in a shared repository at regular intervals. Every commit made in the repository is then built. This allows the development teams to detect the problems early.

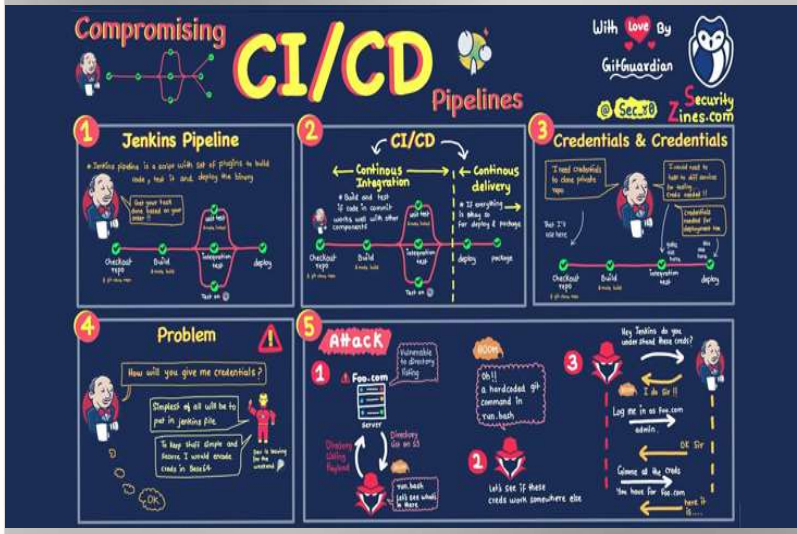
- Continuous integration requires the developers to have regular builds. The general practice is that whenever a code commit occurs, a build should be triggered.

.



Benefits of CI

- **Faster Feedback**: With CI, developers get immediate feedback on whether their changes integrate smoothly with the existing codebase. This helps catch errors early when they're easier and cheaper to fix.
- **Reduced Integration Issues**: By integrating code changes frequently, CI reduces the chances of conflicting changes and integration issues down the line.
- **Higher Quality Code**: Running automated tests as part of the CI process ensures that the code meets quality standards and functions as expected.
- **Increased Confidence**: CI builds confidence among team members and stakeholders by providing a consistent and reliable process for integrating and testing code changes.



Jenkins Continuous Integration

1. **Code Changes**

   Developers commit code changes to the version control system.

2. **Build**

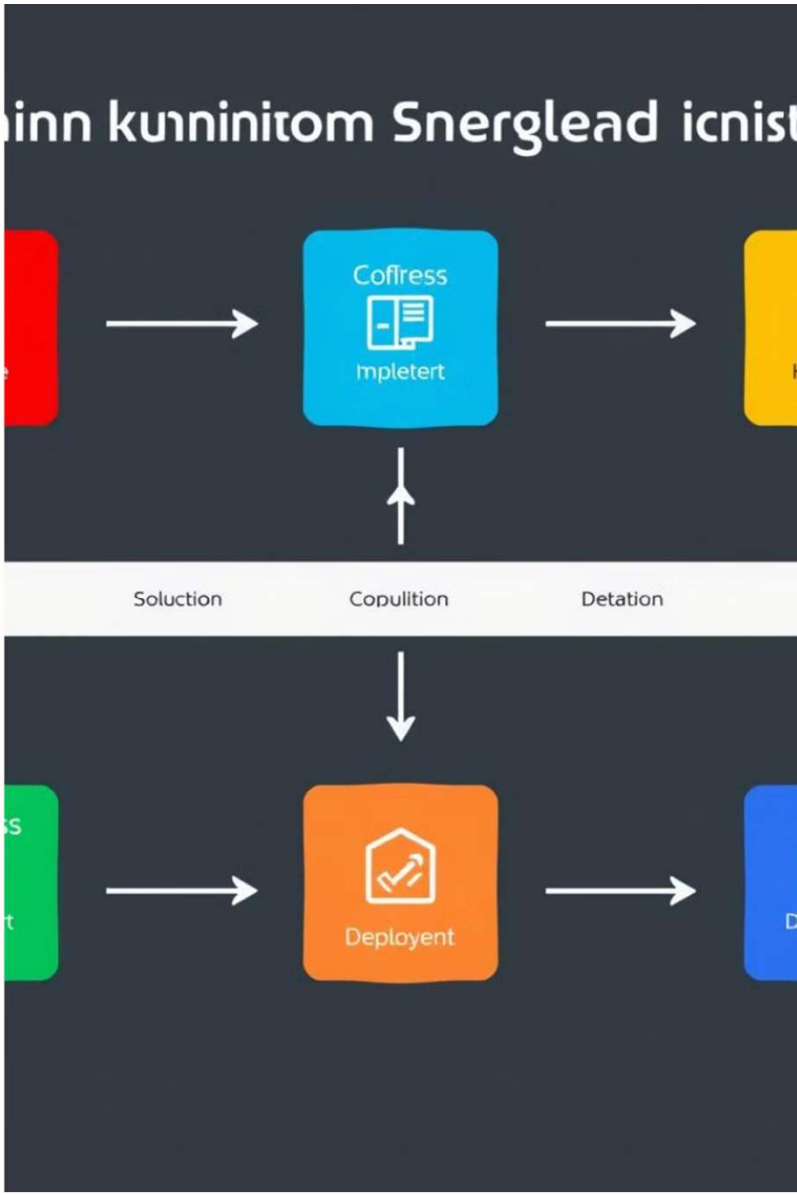
   Jenkins automatically builds the code and runs unit tests.

3. **Test**

   Jenkins runs automated tests, such as integration tests and acceptance tests.

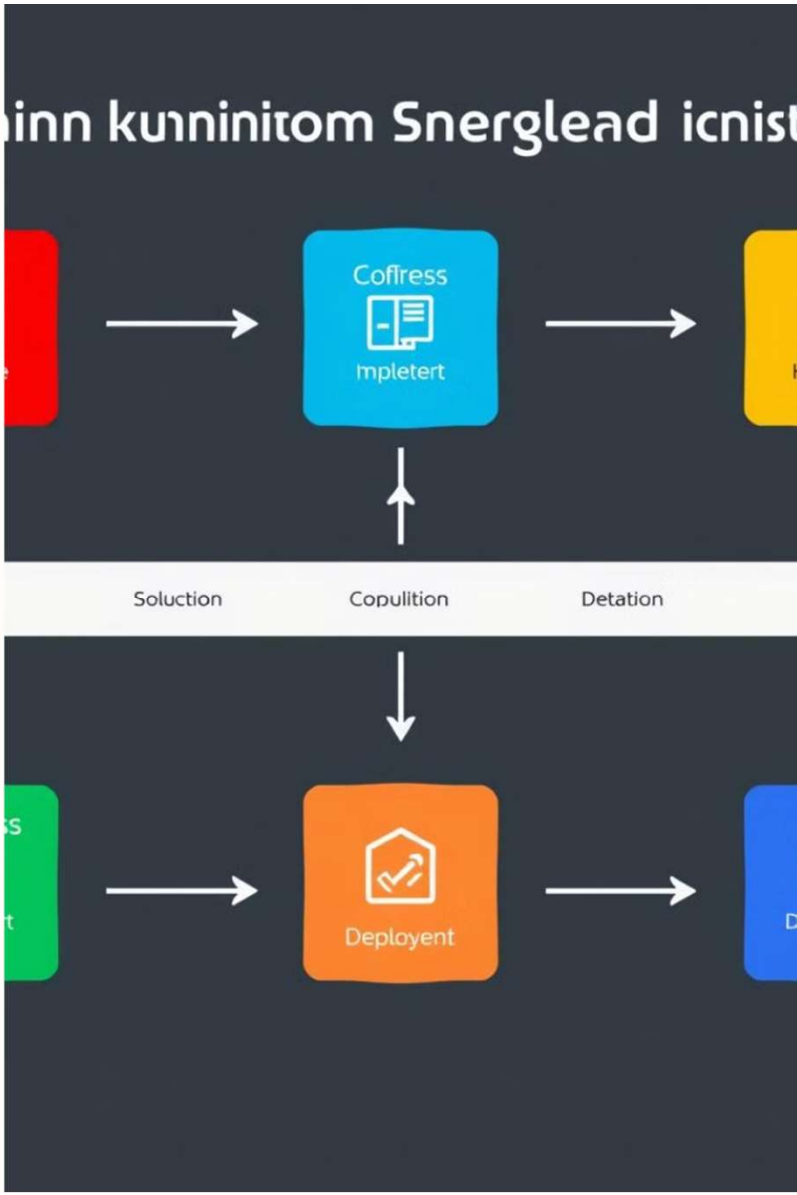
4. **Feedback**

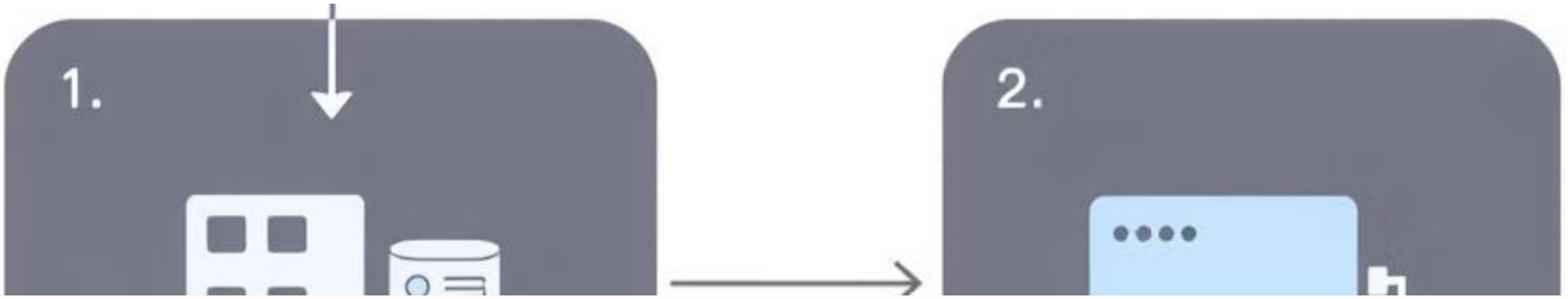
   Jenkins provides immediate feedback to developers about the build and test results.



Best Practices for CI

- **Automate Everything**: Automate the build, test, and deployment processes as much as possible to ensure consistency and reliability.
- **Test Everything:** Write automated tests for all aspects of your application to verify that new changes don't break existing functionality.
- **Commit Frequently**: Encourage developers to commit their changes to the shared repository frequently, ideally several times a day.
- **Monitor Build Status:** Keep an eye on the CI server to monitor the status of builds and address any failures promptly.





Jenkins Continuous Process

1

Continuous Integration

Jenkins automates the build, test, and integration phases of the software development process.

2

Automated Deployment

Jenkins automatically deploys the tested code to the desired environment, whether it's staging or production.

3

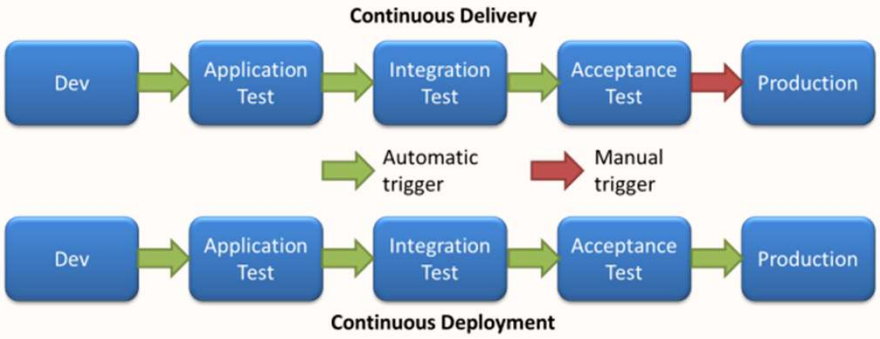
Continuous Delivery

Jenkins ensures that the software is always ready to be deployed, allowing for frequent updates and releases.

Continuous Delivery

Continuous Delivery ensures that your codebase is always in a deployable state and that software can be released into production at any time with minimal manual intervention. However, the final release into production is still a manual step.

- Code changes are automatically tested and prepared for release to production.
- Release to production requires **manual approval** or a final manual step.
- Focuses on keeping the codebase in a **ready-to-deploy** state.
- Ensures that any build passing the pipeline can be released, but it is the team or stakeholder's decision when to deploy.
- Helps ensure reliability, as every stage leading up to deployment is automated and tested.

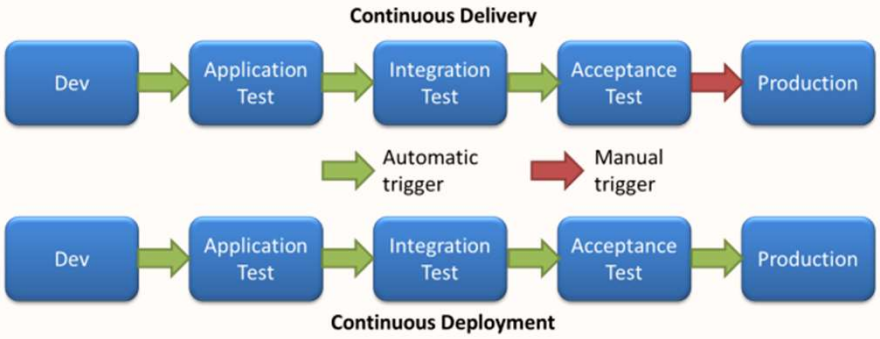


Continuous Deployment

Continuous Deployment takes Continuous Delivery a step further by automating the entire process, including the deployment to production. Every change that passes the automated tests is immediately deployed to production **without manual intervention**.

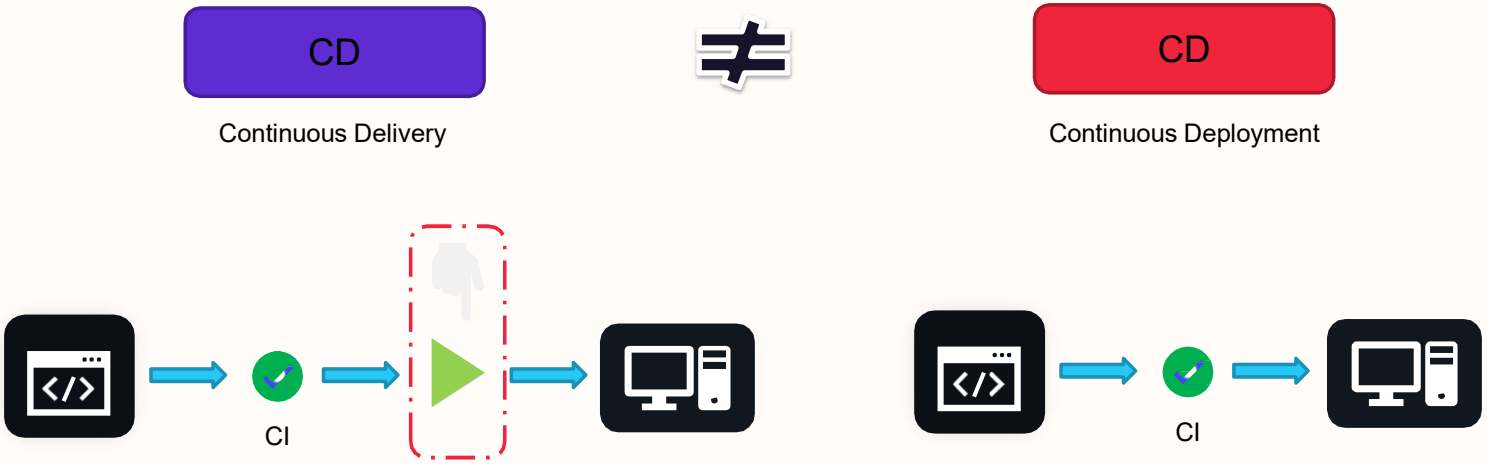
Key Features:

- **No manual approval** for deployment to production.
- Every successful build that passes the test suite is automatically pushed to production.
- Requires a highly reliable and robust automated testing framework since there's no human gatekeeping.
- Encourages frequent, small updates, enabling faster feedback from users and reduced risk.
- Ensures quicker delivery of features, bug fixes, and improvements to customers.



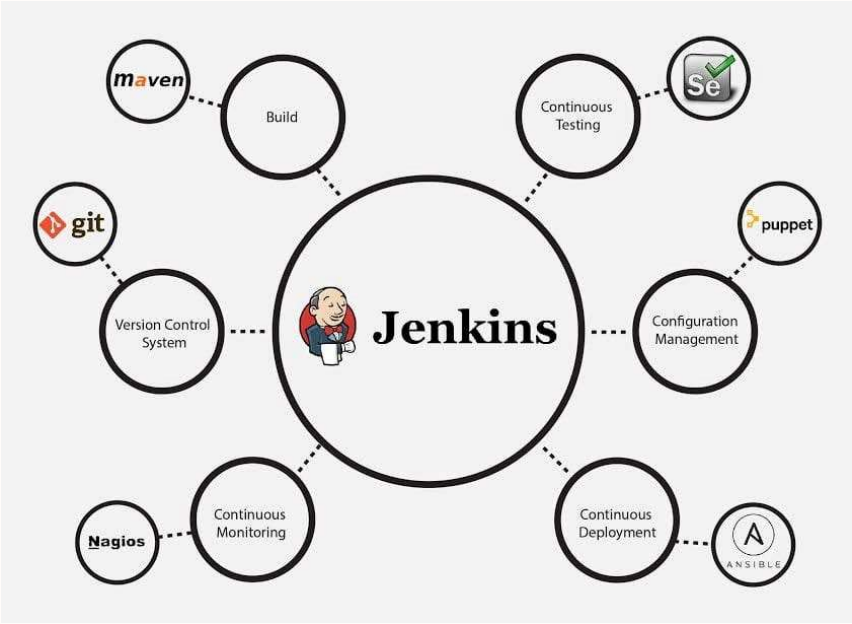
Continuous Deployment vs Delivery

The basic difference between Continuous Delivery and Continuous Deployment is that in Continuous Delivery to deploy the code after the CI process you have to manually trigger it via some button to deploy on the system whereas in Continuous Deployment this process is automatic.



What is Jenkins?

Jenkins is an open-source automation server that helps to automate tasks related to software development. It is popular for its versatility and user-friendliness, making it an ideal choice for organizations of all sizes.



What is Jenkins?

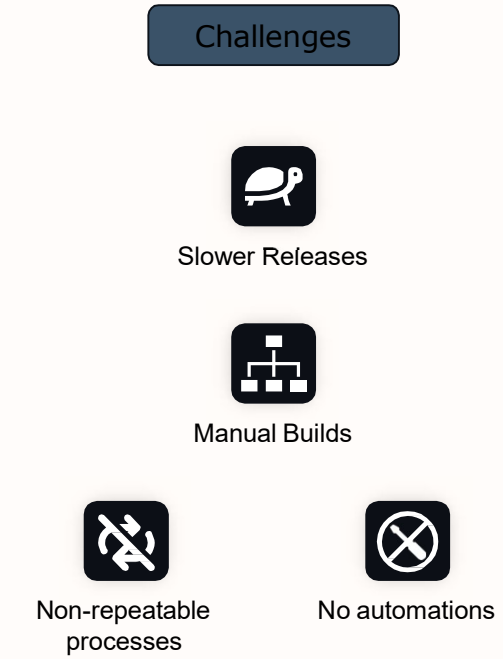
- Jenkins is an open-source automation tool written in Java with plugins built for Continuous Integration purposes.
- Jenkins is used to building and test your software projects continuously making it easier for developers to integrate changes to the project, and making it easier for users to obtain a fresh build.
- It also allows you to continuously deliver your software by integrating with a large number of testing and deployment technologies.

Before The Jenkins

When working on a project with different teams, developers often face issues with different teams using different CI tools, version management, and other tools.

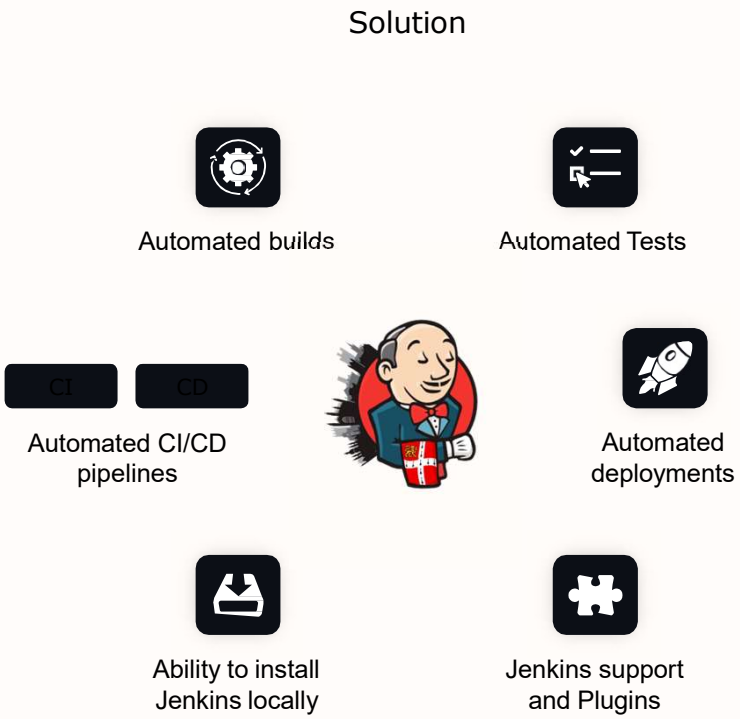
Setting up a CI/CD toolchain for each new project will lead to certain challenges like:

- Slower Releases
- Manual Builds
- Non-repeatable processes
- No Automations



Why Jenkins?

- **Automation**: Automates repetitive tasks in software development.
- **Integration**: Supports various tools and technologies like Git, Docker, AWS, Maven, etc.
- **Extensibility**: Hundreds of plugins available for integration with multiple tools.
- **Ease of Use**: User-friendly web interface with real-time updates and reports.



Open-source

1000+ plugins

Free

Paid, Enterprise

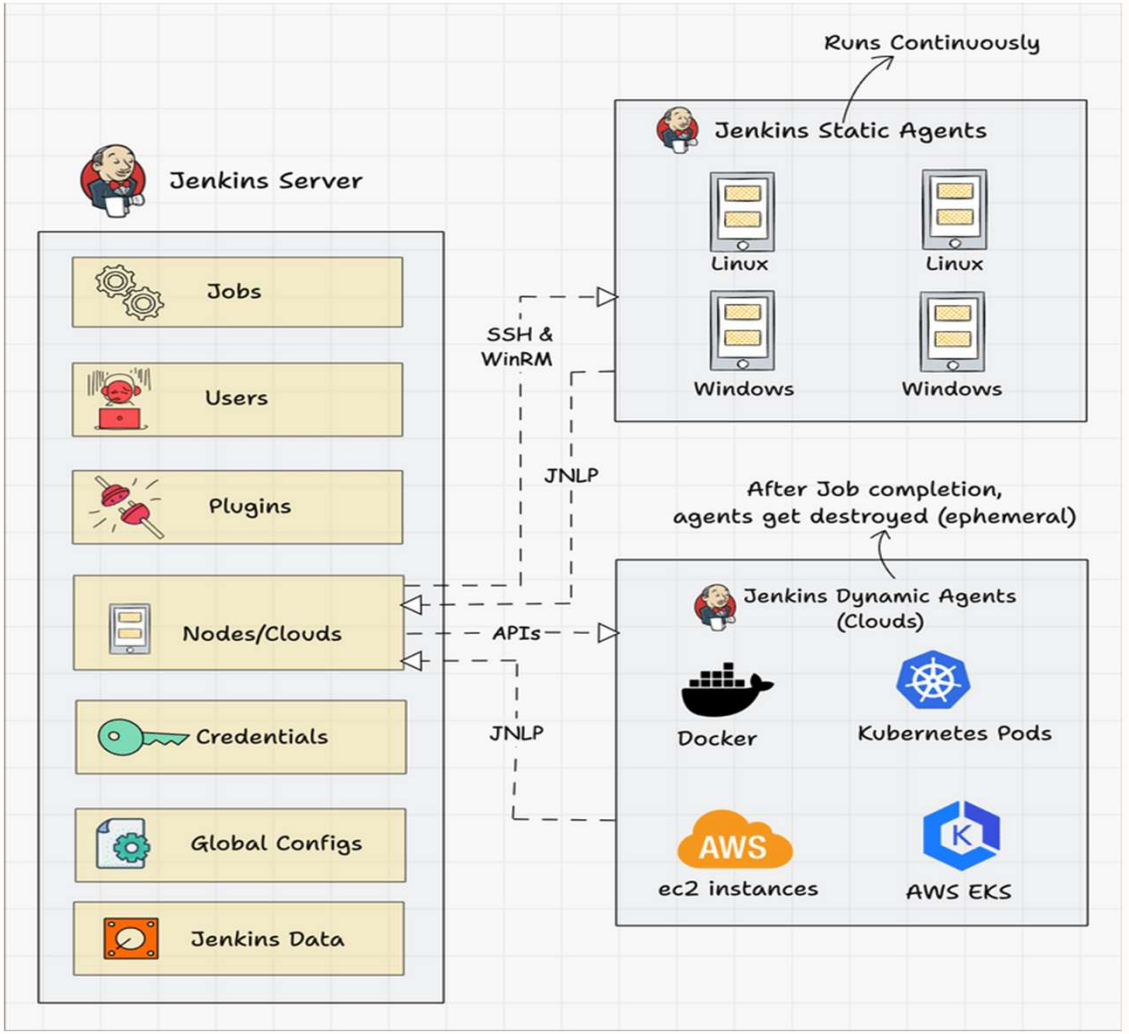
Why Jenkins

- It is open source so it’s free
- It is user-friendly, easy to install and you can easily install additional components. Ready to use in 5 minutes
- Easy to implement new Jenkins plugins for your purposes but most plugins on the market will help you achieve your problems. It has various plugins which make Jenkins flexible.
- No platform dependency. Jenkins can run on any platforms, whether it’s OS X, Windows or Linux.
- It supports 1000 or more plugins to ease your work. If a plugin does not exist, you can write the script for it and share with community.

Jenkins Architecture

Following are the key components in Jenkins

- Jenkins Master Node
- Jenkins Agent Nodes/Clouds
- Jenkins Web Interface



Jenkins Architecture

Master

Jenkins's server or master node holds all key configurations. Jenkins master server is like a control server that orchestrates all the workflow defined in the pipelines. For example, scheduling a job, monitoring the jobs, etc.

Agents

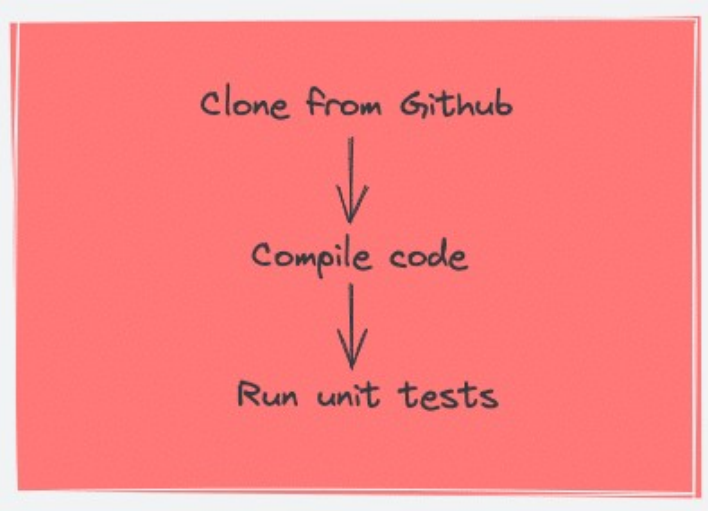
Workers that perform tasks such as building, testing, and deploying software. Agents can run on different machines and operating systems.

Plugins

Extensions that provide additional functionality to Jenkins, such as support for different version control systems, build tools, and deployment methods.

Jenkins Jobs

A job is a collection of steps that you can use to build your source code, test your code, run a shell script, run an Ansible role in a remote host or execute a terraform play, etc. We normally call it a Jenkins pipeline.




```
        stage('Code Checkout') {
            steps {
                checkout([
                    $class: 'GitSCM',
                    branches: [[name: '*/master']],
                    userRemoteConfigs: [[url: 'https://github.com/spring-
projects/spring-petclinic.git']]
                ])
            }
        }

        stage('Code Build') {
            steps {
                 sh 'mvn install -Dmaven.test.skip=true'
            }
        }
```

Types of Jenkins Jobs

Pipeline Jobs

These are defined as code and allow for complex workflows.

Freestyle Jobs

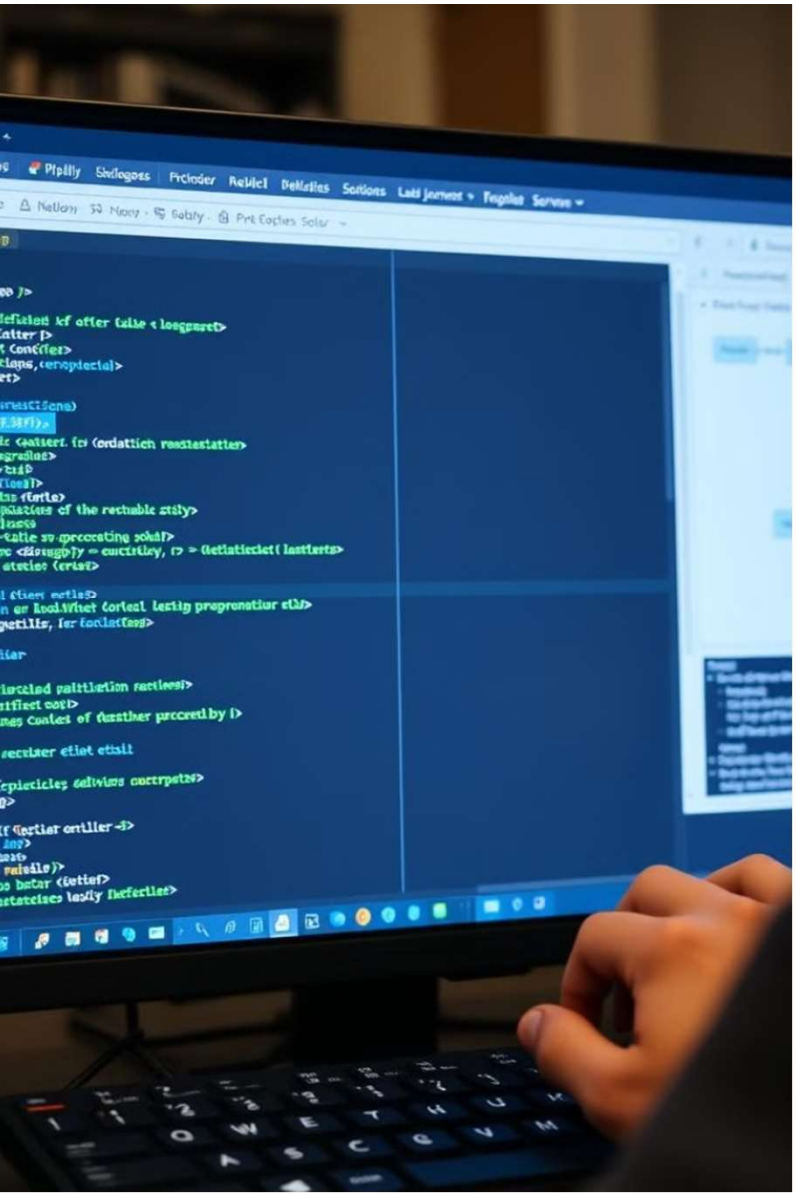
These are simpler, configuration-based jobs.

Multibranch Pipeline Jobs

These run pipelines for each branch of a repository.

Build Triagers

These analyze and categorize builds based on their status.



Pipeline Jobs

1 **Declarative Pipelines**

Easy to read and write. They are defined using a DSL (Domain Specific Language).

2 **Scripted Pipelines**

More powerful and flexible, but require more coding experience.

3 **Blue Ocean**

A user interface that provides a visual representation of the pipeline workflow.

4 **Jenkinsfile**

A file that defines the pipeline and its steps.

Freestyle Jobs

Build Triggers

Triggers can be scheduled, based on SCM changes, or manually.

Build Steps

Steps can include compiling code, running tests, and deploying artifacts.

Post-Build Actions

These are actions that are performed after the build, such as sending notifications or archiving artifacts.

Multibranch Pipeline Jobs

1

Automatic Branch Discovery

New branches are detected and automatically added as pipelines.

2

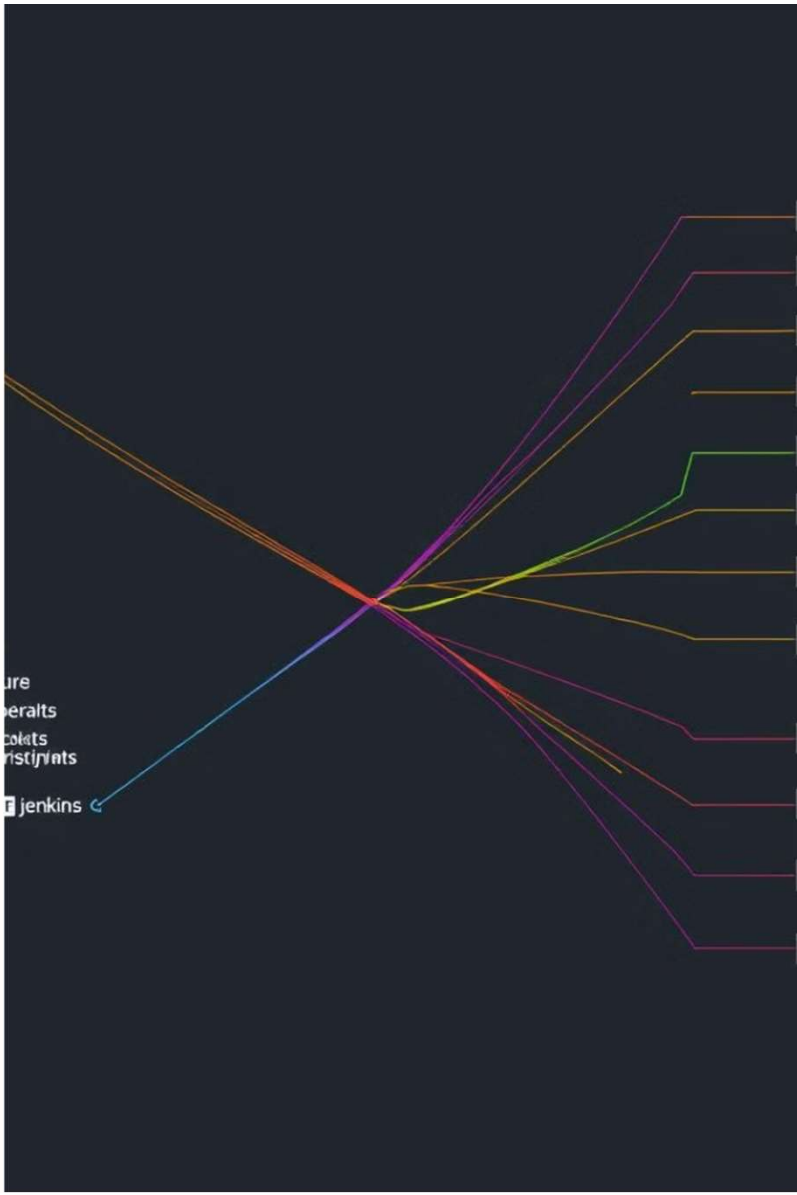
Individual Pipelines

Each branch has its own pipeline, allowing for parallel execution.

3

Pipeline Configuration

Shared pipelines can be defined for all branches with variations per branch.



Build Triagers

In Jenkins, a build trigger is an event or condition that initiates the execution of a Jenkins job. Build triggers enable automation by automatically starting a job based on changes in the source code repository, scheduled time, external events, or other factors.

These are the most common Jenkins build triggers:

1. Trigger builds remotely
2. Build after other projects are built
3. Build periodically
4. GitHub webhook trigger for GITScm polling
5. Poll SCM

Trigger builds remotely

1

Remote Trigger Plugin

This plugin allows you to trigger builds from other systems.

2

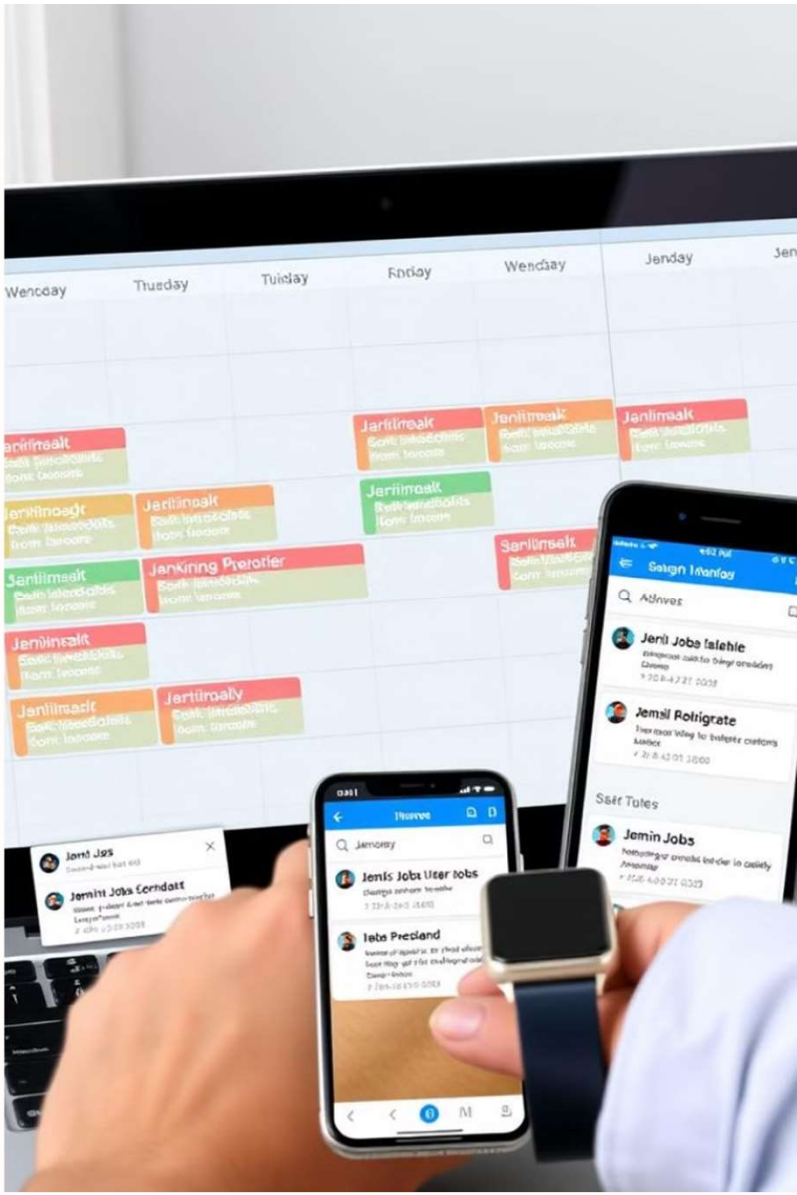
Scheduled Builds

Allows you to schedule builds to run at specific times or intervals.

3

Cron Syntax

Use cron syntax to specify the timing of scheduled builds.



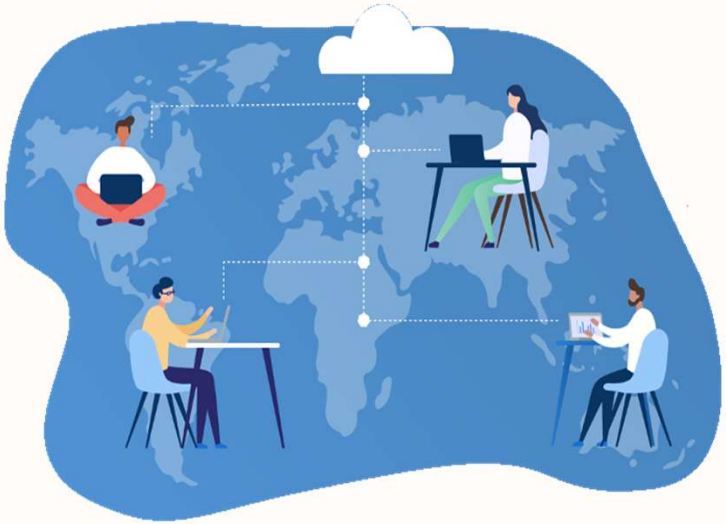
Trigger builds remotely

Description: This trigger allows you to start a Jenkins build remotely by accessing a specific URL with a token. When a request is made to this URL with the correct token, Jenkins initiates a build for the associated job.

Configuration: In the job configuration settings, you enable the "Trigger builds remotely" option and specify a token. Jenkins then generates a URL with the token appended, which can be used to trigger builds remotely.

Use Case: This trigger is useful when you need to start Jenkins builds from external scripts, applications, or services. For example, you can trigger builds from your CI/CD pipeline scripts, deployment automation tools, or even from a simple curl command.

https:/admin:admin@jenkinsserver.job.token

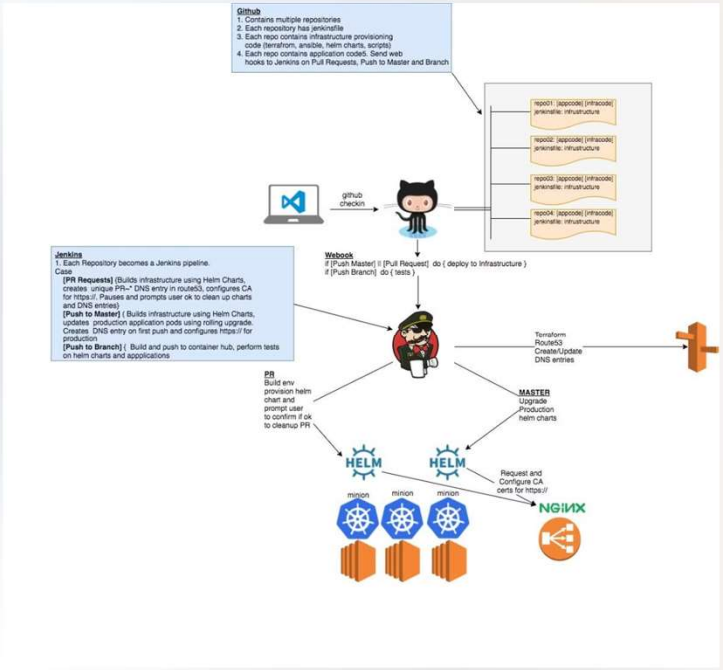


Build Periodically

Description: This trigger allows you to schedule Jenkins builds to run at specific times or on a recurring schedule. You can define the schedule using cron syntax or a simpler interface provided by Jenkins.

Configuration: In the job configuration settings, you specify the schedule for the build using either cron syntax or the Jenkins scheduling interface. Jenkins will automatically trigger builds according to the defined schedule.

Use Case: This trigger is useful for running tasks such as nightly builds, daily tests, or periodic deployments. It provides a way to automate repetitive tasks on a regular basis without manual intervention.



Build after other projects are built

Description: This trigger allows you to configure a Jenkins job to automatically start a build after one or more other projects have been built successfully. It establishes a dependency relationship between jobs.

Configuration: In the job configuration settings, you specify the upstream projects that trigger the current job. When the specified upstream projects complete successfully, Jenkins automatically triggers the current job.

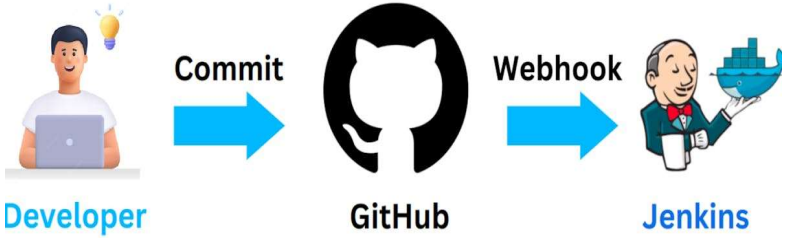
Use Case: This trigger is commonly used for setting up build pipelines or workflows where the completion of one stage triggers the start of another stage. For example, you can have a build job trigger a test job, and the test job trigger a deployment job upon successful completion.

GitHub webhook trigger for GITScm polling

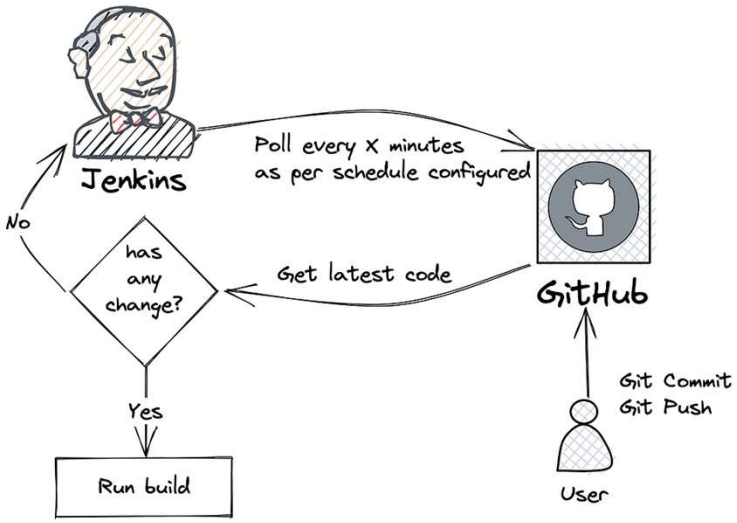
Description: This trigger allows Jenkins to receive notifications from GitHub repositories via webhook requests. When a push event occurs in the repository, GitHub sends a webhook notification to Jenkins, prompting it to poll the repository for changes using Git SCM polling.

Configuration: To set up this trigger, you configure the GitHub repository to send webhook notifications to Jenkins whenever a push event occurs. In the Jenkins job configuration settings, you enable Git SCM polling to check for changes triggered by the webhook.

Use Case: This trigger is ideal for real-time build triggering, as it initiates builds immediately after code changes are pushed to the GitHub repository. It reduces build latency and ensures that Jenkins reacts quickly to changes.



Poll SCM



Description: This trigger allows Jenkins to poll the source code repository for changes at regular intervals. If there are new commits since the last build, Jenkins initiates a new build.

Configuration: In the job configuration settings, you enable SCM polling and specify the polling frequency. Jenkins will check the repository at the specified interval (e.g., every minute, every hour) to detect changes.

Use Case: This trigger is useful when you want to trigger builds automatically whenever there are code changes committed to the repository. It's a simple and effective way to automate the build process in response to code changes.

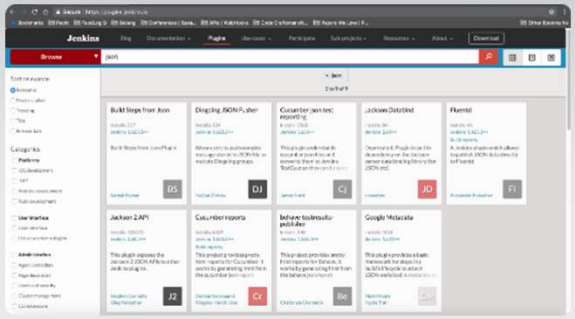
Jenkins Plugins

Plugins are official and community-developed modules that you can install on your Jenkins server. It helps you with more functionalities that are not natively available in Jenkins.

For example, if you want to upload a file to s3 bucket from Jenkins, you can install an AWS Jenkins plugin and use the abstracted plugin functionalities to upload the file rather than writing your own logic in AWS CLI. The plugin takes care of error and exception handling.

```
s3Upload(
    file:'file.txt',
    bucket:'my-bucket',
    path:'path/to/target/file.txt'
    )

s3Upload(
    file:'someFolder',
    bucket:'my-bucket',
    path:'path/to/targetFolder/'
    )
```



Jenkins Plugins

1 Version Control

Plugins for integrating with popular version control systems like Git, SVN, and Mercurial.

2 Build Tools

Plugins for integrating with popular build tools such as Maven, Gradle, and Ant.

3 Deployment

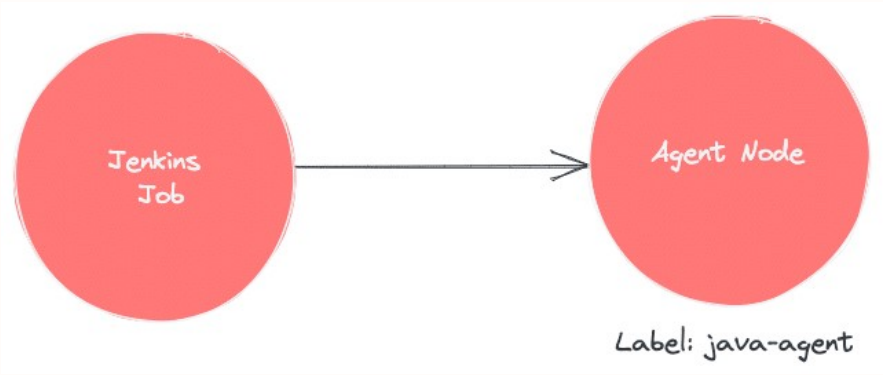
Plugins for deploying software to different environments, such as cloud platforms, on-premise servers, and mobile devices.

4 Notifications

Plugins for sending notifications to users and teams about build status, failures, and other events.

Jenkins Agent

Jenkins Agent



Jenkins agents are the worker nodes that actually execute all the steps mentioned in a Job. When you create a Jenkins job, you have to assign an agent to it. Every agent has a label as a unique identifier.

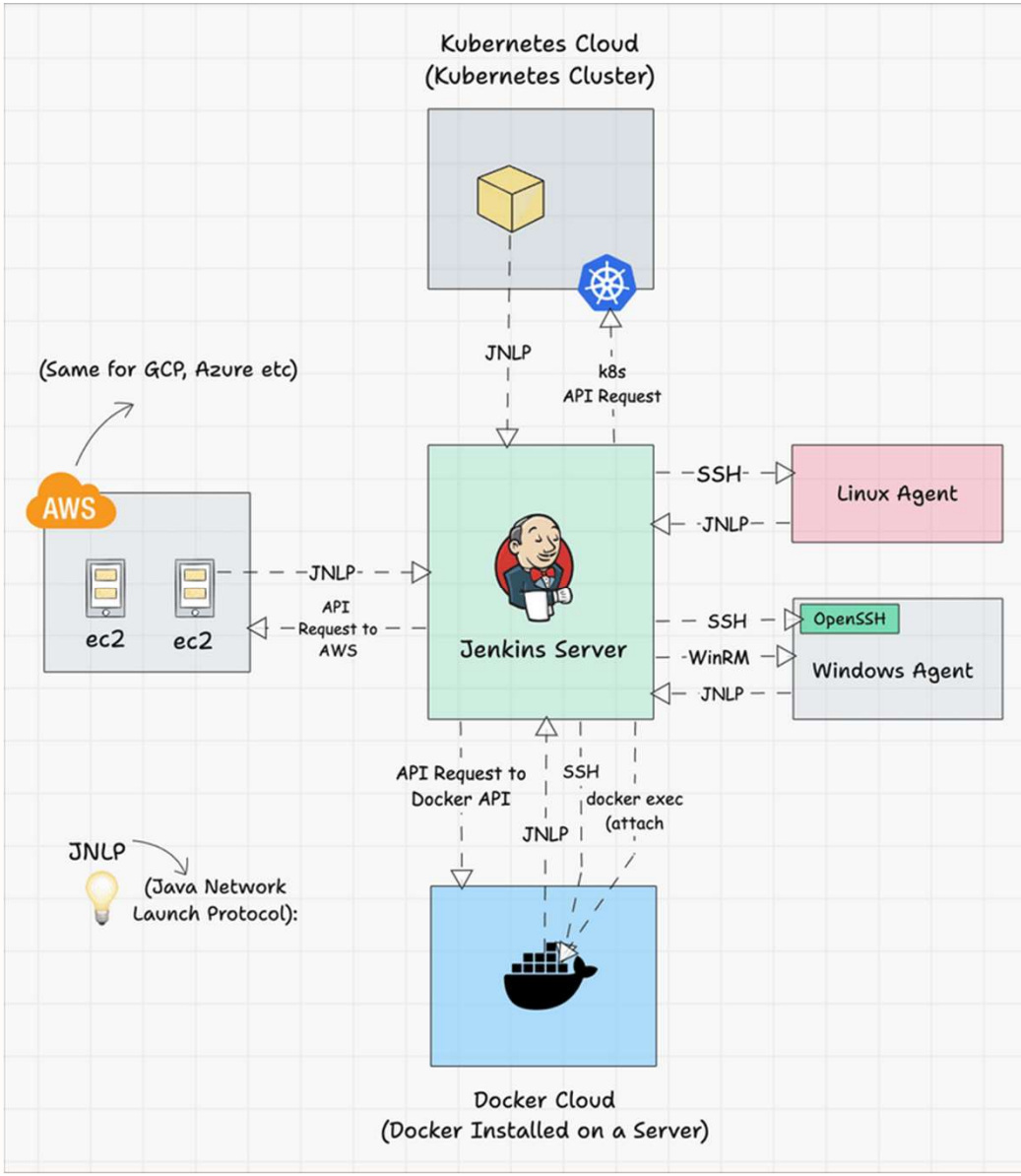
Jenkins Agent Types

1. **Agent Nodes:** These are servers (Windows/Linux) that will be configured as static agents. These agents will be up and running all the time and stay connected to the Jenkins server. Organizations use custom scripts to shut down and restart the agents when is not used. Typically during nights & weekends.
2. **Agent Clouds:** Jenkins Cloud agent is a concept of having dynamic agents. Means, whenever you trigger a job, a agent gets deployed as a VM/container on demand and gets deleted once the job is completed. This method saves money in terms of infra cost when you have a huge Jenkins ecosystem and continuous builds.

Jenkins server-agent Connectivity

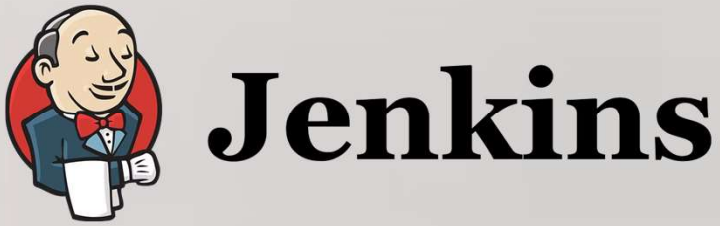
You can connect a Jenkins master and agent in two ways

1. **Using the SSH method:** Uses the ssh protocol to connect to the agent. The connection gets initiated from the Jenkins master. Ther should be connectivity over port 22 between master and agent.
2. **Using the JNLP method:** Uses java JNLP protocol (Java Network Launch Protocol). In this method, a java agent gets initiated from the agent with Jenkins master details. For this, the master nodes firewall should allow connectivity on specified JNLP port. Typically the port assigned will be 50000. This value is configurable.



Jenkins Pipeline: A Powerful Automation Framework

Jenkins Pipeline empowers automated software development workflows, enabling continuous integration and delivery. By defining pipelines in Jenkinsfiles, teams can streamline processes, manage deployments, and enhance efficiency.



What is Jenkins Pipeline?

Automated Workflows

Jenkins Pipeline defines a series of automated steps for software builds, tests, and deployments.

Declarative Syntax

A readable and structured way to define pipeline steps using a simple, intuitive language.

Version Control Integration

Pipeline definitions stored in source control ensure reproducibility and traceability.

Understanding the Jenkins Pipeline Syntax

1 Stages

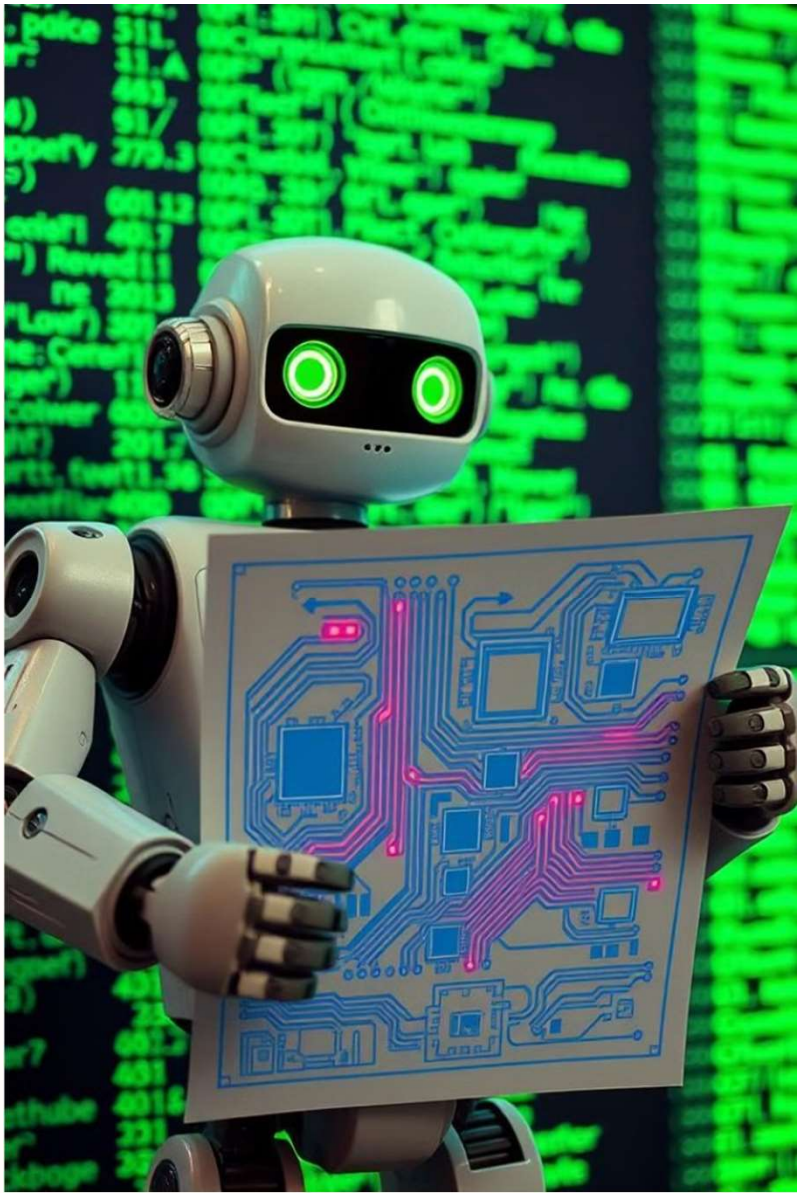
Group related tasks into logical stages (e.g., build, test, deploy).

2 Steps

Individual actions within a stage, like building a project or running tests.

3 Variables

Store and manipulate data throughout the pipeline for flexibility and reusability.



Creating a Jenkins File

1

Define Pipeline Structure

Organize stages and steps in a clear and logical sequence.

2

Specify Execution Environment

Define the tools, resources, and settings required for pipeline execution.

3

Version Control Integration

Store the Jenkinsfile in a Git repository alongside the codebase.

Defining Stages in a Jenkins File

1. **Build Stage**

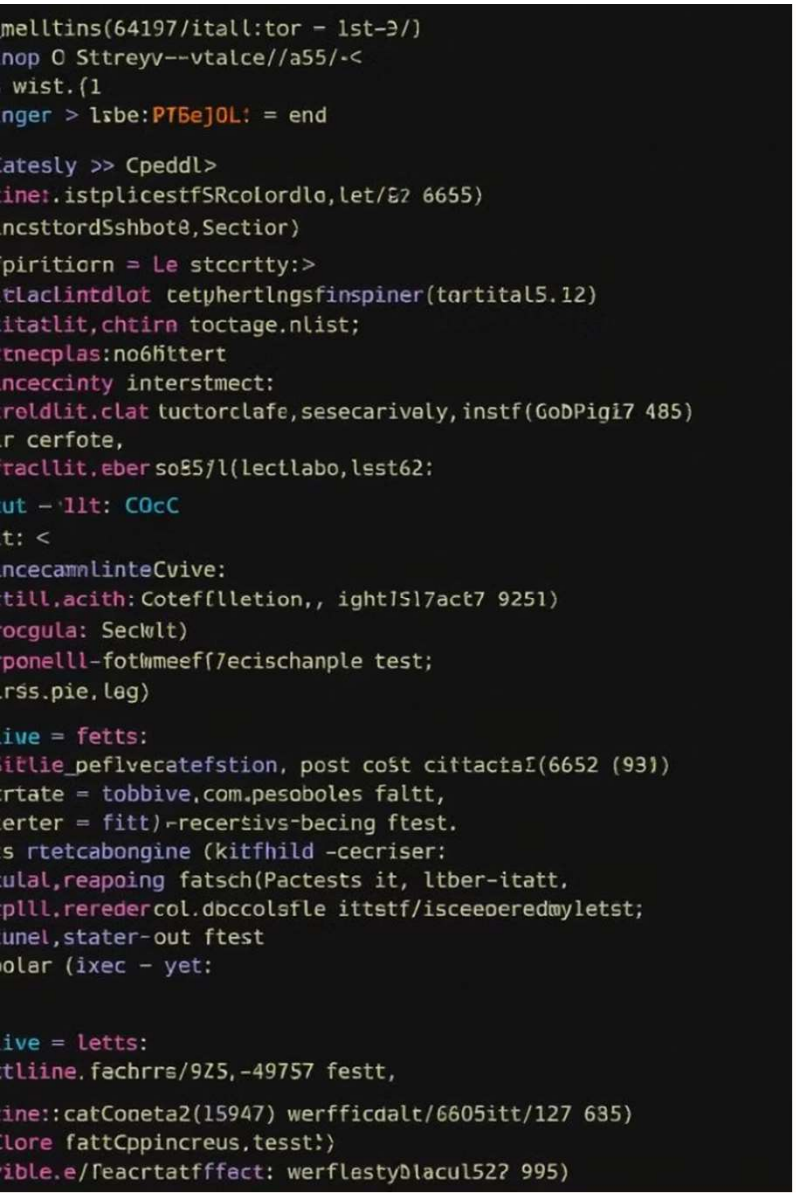
   Compile the source code and create a build artifact.

2. **Test Stage**

   Run unit tests, integration tests, and other automated checks.

3. **Deploy Stage**

   Deploy the build artifact to a target environment (e.g., development, production).



Jenkins Scripts

Groovy

Jenkins uses Groovy, a Java-based scripting language, to define and configure jobs and pipelines.

Pipeline as Code

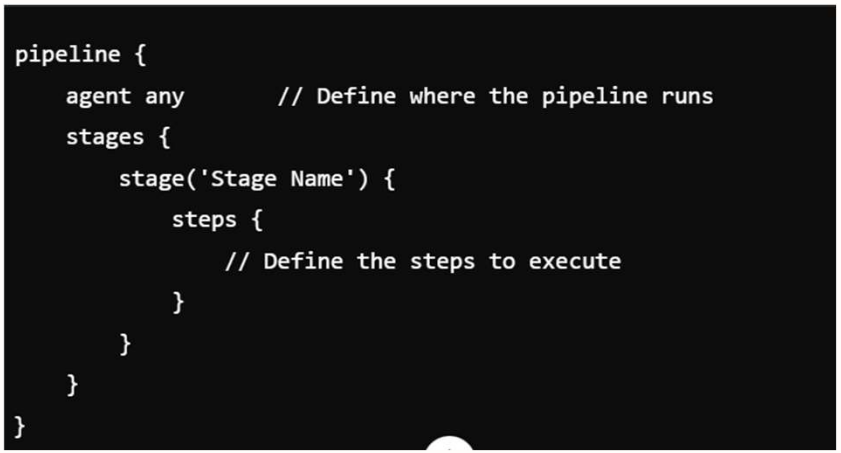
Jenkins allows you to define and manage your CI/CD pipelines as code, ensuring consistency and reproducibility.

Flexibility and Extensibility

Scripts provide a flexible and powerful way to customize Jenkins workflows and integrate with other tools and services.

Jenkins File Scripts

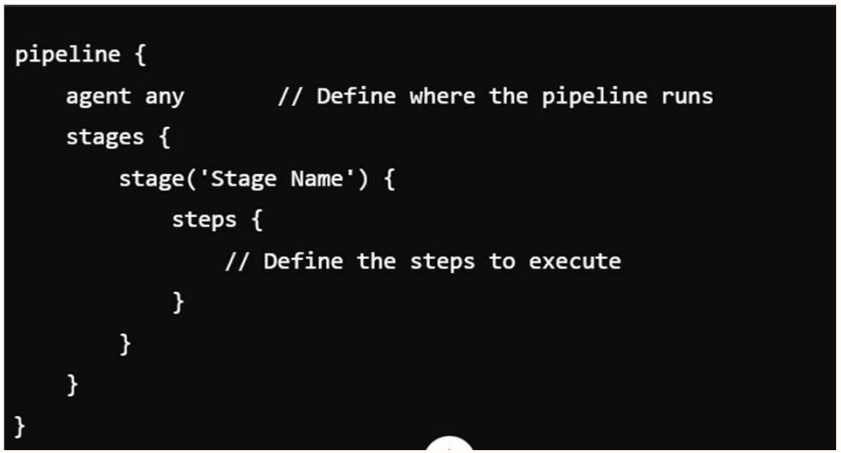
A **Jenkins file** is a text file that contains the definition of a Jenkins Pipeline, typically written using the **Groovy** language. There are two types of Jenkins Pipelines: **Declarative Pipeline** and **Scripted Pipeline**.



```
pipeline {
    agent any       // Define where the pipeline runs
    stages {
        stage('Stage Name') {
            steps {
                // Define the steps to execute
            }
        }
    }
}
```

Jenkins File Scripts

A **Jenkins file** is a text file that contains the definition of a Jenkins Pipeline, typically written using the **Groovy** language. There are two types of Jenkins Pipelines: **Declarative Pipeline** and **Scripted Pipeline**.



```
pipeline {
    agent any       // Define where the pipeline runs
    stages {
        stage('Stage Name') {
            steps {
                // Define the steps to execute
            }
        }
    }
}
```

Parallel Execution in Jenkins Pipeline

Parallel Stage Execution	Run stages concurrently for faster execution.
Parallel Step Execution	Execute multiple steps within a stage in parallel for improved efficiency.
Parallel Workflow	Break down complex tasks into smaller, parallel workflows for increased throughput.

Triggering Jenkins Pipelines

1. ## Manual Trigger

   Initiate pipeline execution manually from the Jenkins interface.

2. ## Scheduled Trigger

   Define a schedule for automatic pipeline execution at regular intervals.

3. ## Webhooks

   Trigger pipelines automatically when events occur in external systems (e.g., Git push).

Best Practices and Troubleshooting

1

Code Reusability

Embrace shared libraries and modular design for efficient pipeline development.

2

Clear Error Handling

Implement robust error handling and logging to facilitate troubleshooting.

3

Continuous Improvement

Regularly review and refine pipelines to optimize performance and efficiency.