SOFTWARE ENGINEERING

UNIT - 4

TOPIC – 1

A STRATEGIC APPROACH TO SOFTWARE TESTING

INTRODUCTION TO TESTING

Software testing is a key step in creating software, ensuring it works correctly and meets user expectations. Think of it as quality control for a product before it reaches customers.

What is Software Testing?

Software testing checks:

1. **Does it work as expected?** Ensures the software produces the correct results.
2. **Does it meet user needs?** Verifies that the software fulfills the requirements it was built for.
3. **Is it free of bugs?** Identifies and fixes errors before users encounter them.

Why is Testing Important?

- Prevents failures in important tasks (e.g., bank transfers or online shopping).
- Ensures the software runs smoothly under all conditions.
- Protects sensitive data from security threats.

Since errors can disrupt the user experience and lead to significant consequences, testing is necessary to catch these issues before the software is deployed. However, finding these errors is only one part of the solution. Once identified, errors must be fixed through debugging.

TESTING VS DEBUGGING

- **Testing**: Finds problems by examining how the software behaves under different conditions.

  - Example: Testing a login page to check if it accepts correct credentials and rejects incorrect ones.
- **Debugging**: Fixes the problems found during testing.
  - Example: Correcting the login page so it stops crashing when wrong credentials are entered.

Aspect	Testing	Debugging
Who Does It?	Testers or Quality Analysts	Developers
Focus	Finds problems in the software	Fixes what caused the problems
Question Asked	Does it work?	Why isn't it working, and how to fix it?

Testing sets the stage for debugging by identifying where the software fails. To structure the testing process efficiently, we use specific software testing strategies.

A STRATEGIC APPROACH TO SOFTWARE TESTING

A **testing strategy** is a systematic plan to ensure all parts of the software are tested effectively. It provides a step-by-step approach to check individual components, their integration, and the entire system.

Goals of Testing Strategies

1. Identify and fix errors early to save time and resources.
2. Ensure the software performs as expected under different conditions.
3. Verify that the software meets both functional and user requirements.

Types of Testing Strategies

1. **Unit Testing**: The first step of testing, focusing on individual components of the software.
   - Example: Testing a login function with valid and invalid inputs to ensure it works correctly.
2. **Integration Testing**: After unit testing, integration testing ensures that different components work together.

   - Example: Checking if the login system successfully connects to the user dashboard.
3. **System Testing**: Tests the entire software as a single system to ensure all parts work together seamlessly.
   - Example: Testing a food delivery app to ensure login, menu selection, and payment features work correctly.
4. **Validation Testing**: Focuses on ensuring the software meets the end user's needs.
   - Example: Allowing real users to test a beta version of the app to provide feedback.

With these strategies, testing progresses logically, starting with small units and expanding to the complete system. Let's now explore these types of testing in more detail.

Black-Box and White-Box Testing

Black-Box Testing

Black-Box Testing focuses on testing the software's functionality without looking at the internal code. It examines inputs and outputs to check if the software behaves as expected.

- **Key Features**:
  - Tester doesn't need programming knowledge.
  - Based on user requirements and specifications.
- **Examples**:
  - Testing a calculator app by entering numbers and verifying results.
  - Checking if an e-commerce website correctly adds items to the cart.

White-Box Testing

White-Box Testing tests the internal structure or logic of the code. It ensures that all possible paths in the code work as intended.

- **Key Features**:
  - Requires programming knowledge.
  - Focuses on code coverage, such as loops, conditions, and paths.
- **Examples**:

- Verifying if a login function correctly checks both username and password.
- Checking the logic of a sorting algorithm to confirm it handles all input cases.

Differences Between Black-Box and White-Box Testing

Aspect	Black-Box Testing	White-Box Testing
Focus	Software functionality	Internal code structure
Knowledge Required	None (only specifications are needed)	Requires programming and code understanding
Approach	Input-output based	Logic and path-based
Example	Testing login functionality from the user's perspective	Checking how the login function processes inputs internally

VERIFICATION AND VALIDATION

Verification: "Are we building the product correctly?" It ensures the software matches its design and specifications.

- **Example**: Checking if a login screen has all the fields and buttons specified in the design.

Validation: "Are we building the right product?" It ensures the software meets user needs.

- **Example**: Testing if users can successfully log in with the "Forgot Password" feature.

Aspect	Verification	Validation
Focus	Checks if software matches the plan	Checks if software meets user needs
Example	Code reviews	Functional and user testing

Testing ensures software is error-free, but when errors are found, the next step is to debug and resolve them.

SOFTWARE ENGINEERING

UNIT - 4

TOPIC – 2

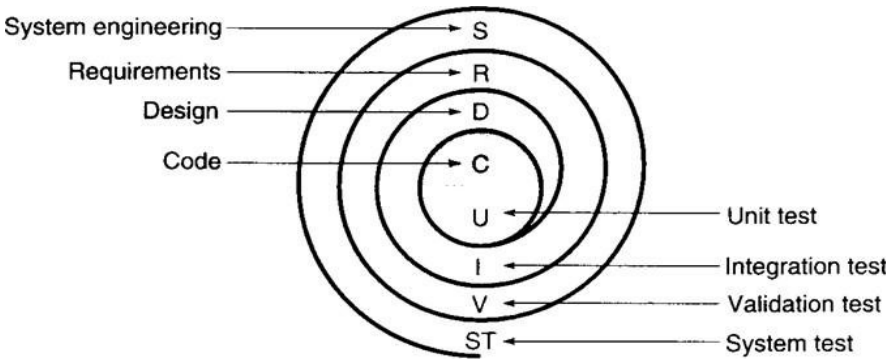
TESTING STRATEGY AND ART OF DEBUGGING

SOFTWARE TESTING STRATEGY FOR CONVENTIONAL SOFTWARE:

A software testing strategy for conventional software ensures the product is high-quality, works as expected, and meets requirements. The process is systematic and follows these steps:

1. **Understand Requirements:** First, the team carefully reviews the software requirements to know exactly what needs to be tested.
2. **Create a Test Plan:** A plan is prepared, explaining what to test, how to test, and the resources needed, including time and people.
3. **Design Test Cases:** Detailed test cases are created to check all possible scenarios, including normal and unusual situations.
4. **Set Up a Test Environment:** A setup is created that is similar to the actual environment where the software will run.
5. **Execute Tests:** Different types of testing, such as checking functionality, performance, and security, are performed to find any issues.
6. **Track and Fix Issues:** Any defects found are recorded, fixed, and tested again to ensure the fixes work and don't create new problems.
7. **Do User Testing:** The software is tested by real users to ensure it meets their needs and works as expected in real-life situations.
8. **Document Everything:** All test plans, test cases, and results are recorded for future reference.
9. **Create a Summary Report:** A report is prepared at the end to summarize the testing activities and results.
10. **Focus on Improvement:** Feedback is gathered to improve the testing process for future projects.

A strategy for software testing may be viewed in the context of the spiral as shown below

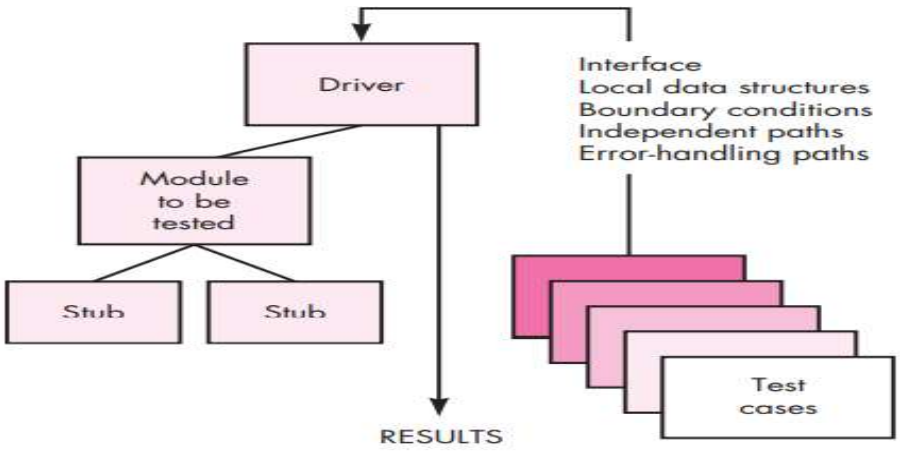


1. Unit Testing

Unit testing is the first step in software testing, focusing on checking individual pieces or "units" of the software in isolation. It's like testing each brick before constructing a wall. This process ensures that every small part of the software works correctly on its own, helping to catch errors early and making them easier and less expensive to fix.

Tools Used in Unit Testing

In software development, different parts of the system often rely on one another. During unit testing, some of these parts may not be ready. To address this, two tools—**drivers** and **stubs**—are used. These tools allow testing of individual components without needing the entire system to be complete.



1. **Drivers**

   Drivers are small programs that trigger and control the testing of specific software parts. They act as substitutes for the part of the system that normally calls the function being tested.

   - **Example**: Consider testing the login function of a website. Normally, the function is activated when a user clicks the "Login" button. Instead of waiting for the complete user interface to be ready, a driver directly calls the login function with sample inputs (e.g., a username and password) to check if it works properly.

2. **Stubs**

   Stubs are placeholders for incomplete or missing parts of the system. They simulate the behavior of the unavailable components by providing fake responses.

   - **Example**: Think about testing an online store's checkout system. The checkout might rely on a payment gateway that hasn't been connected yet. A stub can act as the payment gateway and provide fake responses like "Payment Successful" or "Payment Failed," allowing the checkout system to be tested without the actual gateway.

How Drivers and Stubs Work Together

Drivers and stubs work as temporary tools to fill the gaps during testing:

- The **driver** starts the test by sending inputs to the unit being tested.
- If the unit relies on other parts of the software that are not yet available, **stubs** step in to mimic those parts and provide fake responses.

This setup ensures that each component can be tested in isolation, even if the other parts are still under development.

Importance of Unit Testing

- **Speeds Up Testing**: Drivers and stubs enable testing of specific features without waiting for the entire system to be ready.
- **Early Detection of Errors**: Testing smaller parts first helps find and fix problems early.

- **Saves Time and Effort**: Isolated testing avoids delays caused by missing or incomplete components.

Unit testing ensures that the smallest building blocks of software work correctly. Once all units function properly, the next stage is to test how these units interact with one another—this is called **integration testing**.

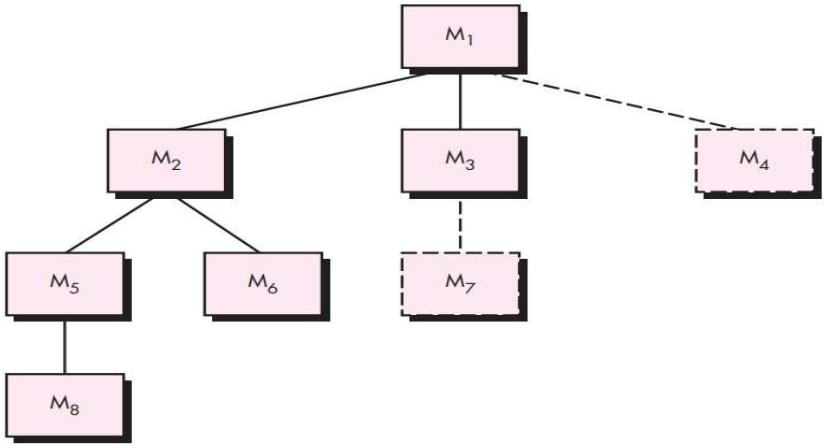
Once individual units are tested, the next step is to test how they interact with one another.

2. Integration Testing

Integration testing ensures that different parts (modules) of the software work together smoothly. After individual units are tested, they are combined, and their interactions are tested to confirm that the system functions as expected when all parts are integrated.

Top-Down Integration Testing

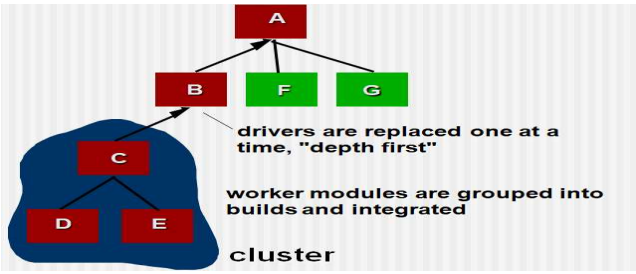
This method begins testing with the main module at the top of the system and progresses downward to the lower modules. Missing or unfinished lower modules are replaced with stubs during testing.



- For example, in a mobile app, the top-level feature, such as the dashboard, is tested first while stubs simulate other sections like notifications or settings.
- In the diagram, testing starts with `M1` at the top, then integrates `M2` and `M3`, while missing modules like `M4` are temporarily replaced with stubs.

Bottom-Up Integration Testing

This method starts testing from the lowest modules and builds upward toward the main module. Higher-level modules not yet available are replaced with drivers that simulate their behavior.



- For instance, in an e-commerce app, lower-level modules like adding items to the cart are tested first, while a driver simulates the checkout process.
- In the diagram, lower modules `D` and `E` are grouped and tested first. Drivers act as temporary placeholders for higher modules like `B` and `A`, progressing step-by-step until the entire system is integrated.

Stubs and Drivers in Integration Testing

- **Stubs** simulate unfinished modules, allowing testing of higher-level components before all parts are ready. For example, a stub can mimic a payment system for testing the checkout feature.
- **Drivers** simulate higher-level modules to test lower-level components. For example, a driver could mimic user interactions to test data-processing modules.

Integration testing ensures that the system works as a whole, whether built from the top down or the bottom up.

After integration testing confirms that components interact correctly, the entire system is tested as a whole.

3. System Testing

System testing checks the entire software system to ensure it works as a whole. It's like inspecting a fully assembled car to ensure all parts function together.

Why System Testing?

- Ensures the software is complete and ready for users.
- Verifies it meets all requirements.

Types of System Testing:

1. **Functional Testing**: Does it do what it's supposed to do?
    - **Example**: Testing if users can add and remove items from their shopping cart.
2. **Performance Testing**: Can it handle heavy use or high traffic?
    - **Example**: Checking if a website can handle 1,000 users at the same time.
3. **Security Testing**: Is it safe from unauthorized access or hacking?
    - **Example**: Testing if passwords are encrypted.
4. **Compatibility Testing**: Does it work on different devices and browsers?
    - **Example**: Checking if a mobile app runs smoothly on both iPhones and Android phones.

4. Validation Testing

Validation testing checks if the software fulfils user needs and works in real-world conditions.

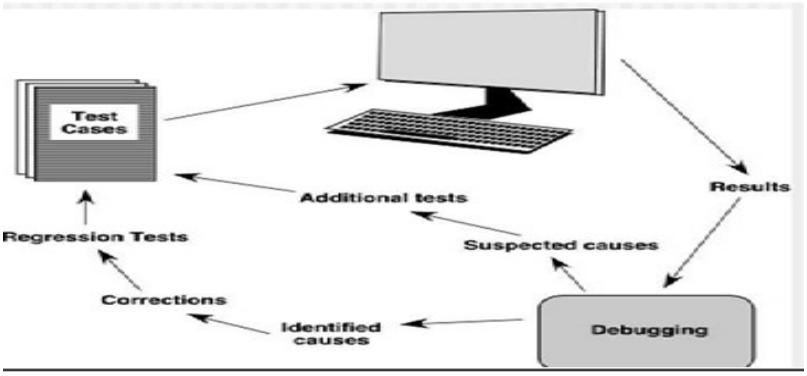
Types of Validation Testing:

1. **Alpha Testing**:
    - Conducted by the development team in a controlled environment.
    - **Example**: Testing a banking app on office computers.
2. **Beta Testing**:
    - Conducted by real users in their own environments.
    - **Example**: Releasing a beta version of an app to collect user feedback.

THE ART OF DEBUGGING

Debugging is a crucial process in software development where developers identify and fix errors (bugs) in the software. It goes beyond just finding the bug; it involves understanding the

root cause of the issue and resolving it without introducing new problems. Debugging ensures the software works as intended and is free from errors.



Steps in Debugging

1. **Identify the Problem**:
    - Understand the issue by reproducing it.
    - Example: If a webpage crashes when submitting a form, try submitting the same form with similar inputs to see the error again.
2. **Locate the Source**:
    - Use tools or manual analysis to find where the problem occurs in the code.
    - Example: If a login form fails, check the function that verifies the username and password.
3. **Analyze the Cause**:
    - Study the code and logic to figure out why the problem happens.
    - Example: A calculation error could be due to incorrect variable initialization or a missing condition in the logic.
4. **Fix the Issue**:
    - Modify the code to resolve the problem.
    - Example: If a variable isn't updated correctly, ensure the correct value is assigned at the right place.
5. **Test the Fix**:
    - Run the software again to confirm the issue is resolved and no new issues are introduced.
    - Example: After fixing a form submission error, test the form with different inputs to ensure it works for all cases.

Types of Debugging / Debugging Strategies

1. **Brute Force Debugging**:
    - Test all possible inputs and conditions until the problem is identified.
    - **Example**: If a function fails for some inputs, feed every possible input into the function until you find which one causes the issue.
2. **Backtracking**:
    - Start at the point where the problem is visible and work backward through the code to find the root cause.
    - **Example**: If a value is incorrect at the end of a calculation, trace each step of the calculation to find where the error occurred.
3. **Cause Elimination**:
    - Systematically disable or isolate parts of the code to pinpoint where the error is happening.
    - **Example**: Temporarily disable different sections of a function to check which section causes the issue.
4. **Print Statement Debugging**:
    - Insert print statements in the code to display variable values or program states at specific points.
    - **Example**: Print the value of a variable inside a loop to see how it changes with each iteration.
5. **Using Debugging Tools**:
    - Tools like debuggers, breakpoints, and logs can simplify the debugging process.
    - **Example**: Use the debugger in an IDE to pause the program at a specific line and inspect all variable values.

Real-Life Example of Debugging

Scenario: A calculator app is giving incorrect results when dividing two numbers.

Steps to Debug:

1. **Identify the Problem**:
    - Observe that the division feature is not working correctly. For example, when dividing 10 by 2, the result is shown as 0 instead of 5.

2. **Locate the Source**:
    - Analyze the code that performs the division. Focus on the function or formula used for the calculation.
3. **Analyze the Cause**:
    - Discover that the division code is written using integer division (`/`) instead of floating-point division, which truncates the result to 0 for non-decimal results.
4. **Fix the Issue**:
    - Modify the code to use the correct division operator (e.g., `float division` instead of `integer division`). Update the function to ensure accurate calculations.
5. **Test the Fix**:
    - Test the calculator by dividing different numbers (e.g., 10/2, 15/3, and 7/2) to ensure the results are correct and accurate (e.g., 5, 5, and 3.5).

SOFTWARE ENGINEERING

UNIT - 4

TOPIC – 3

INSTALLATION OF DOCKER, MINIKUBE, ACCOUNT CREATION IN DOCKERHUB

1. Install Docker

Windows

Step 1: Prepare Your Computer

1. Open **Task Manager** (Ctrl + Shift + Esc).
2. Go to the **Performance** tab.
3. Check if **Virtualization** is enabled. If not:
    - Restart your computer.
    - Enter the BIOS/UEFI settings (press a key like F2, F10, or Del while booting).
    - Enable **Virtualization Technology (VT-x)** or **AMD-V**.

Step 2: Install Docker Desktop

1. Go to Docker's website and download **Docker Desktop for Windows**.
2. Double-click the downloaded file to start the installation.
3. Follow the steps in the installer:
    - Accept the terms.
    - Choose to enable **WSL2** instead of Hyper-V (if prompted).
    - Click **Install** and wait.

Step 3: Start Docker

1. Open **Docker Desktop** (search for it in the Start menu).

2. Sign in with your Docker Hub account (create one at hub.docker.com if you don't have one).

Step 4: Verify Docker is Working

1. Open **PowerShell** (right-click, select **Run as Administrator**).
2. Type this command:

```
docker --version
```

3. You should see the Docker version number. If yes, Docker is installed and running!

Mac

Step 1: Install Docker Desktop

1. Go to Docker's website.
2. Download **Docker Desktop for Mac**.
3. Open the downloaded `.dmg` file.
4. Drag the **Docker** icon into the **Applications** folder.

Step 2: Start Docker

1. Open **Applications** and double-click the **Docker** icon.
2. Docker will start running in the background. Look for the whale icon in the top-right menu bar.

Step 3: Verify Docker is Working

1. Open **Terminal** (search for it in Spotlight using Cmd + Space).
2. Type:

```
docker --version
```

3. You'll see the Docker version number. If yes, Docker is ready!

Linux/Ubuntu

Step 1: Install Docker

1. Open **Terminal**.
2. Update your system:

```
sudo apt update && sudo apt upgrade -y
```

3. Install required tools:

```
sudo apt install apt-transport-https ca-certificates curl software-properties-common -y
```

4. Add Docker's GPG key:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
```

5. Add Docker's repository:

```
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

6. Install Docker:

```
sudo apt update
sudo apt install docker-ce docker-ce-cli containerd.io -y
```

Step 2: Start Docker

1. Start the Docker service:

```
sudo systemctl start docker
```

2. Enable it to run on startup:

```
sudo systemctl enable docker
```

Step 3: Verify Docker is Working

1. Check the version:

```
docker --version
```

2. You'll see the Docker version number if it's running properly.

2. Install Minikube

Windows

Step 1: Download Minikube

1. Open **PowerShell** as Administrator.
2. Download Minikube:

```
curl -LO
https://storage.googleapis.com/minikube/releases/latest/m
inikube-windows-amd64.exe
```

3. Move it to your system path:

```
move minikube-windows-amd64.exe
C:\Windows\System32\minikube.exe
```

Step 2: Start Minikube

1. Start Minikube using Docker:

```
minikube start --driver=docker
```

2. Minikube will set up a Kubernetes cluster using Docker.

Mac

Step 1: Install Minikube

1. Open **Terminal**.
2. Download Minikube:

```
curl -LO
https://storage.googleapis.com/minikube/releases/latest/minikube-darwin-amd64
```

3. Move Minikube to a system path:

```
sudo install minikube-darwin-amd64 /usr/local/bin/minikube
```

Step 2: Start Minikube

1. Run the command:

```
minikube start --driver=docker
```

Linux/Ubuntu

Step 1: Install Minikube

1. Open **Terminal**.
2. Download Minikube:

```
curl -LO
https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
```

3. Move it to a system path:

```
sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

Step 2: Start Minikube

1. Start Minikube with Docker:

   ```
   minikube start --driver=docker
   ```

Common Steps for All Systems

1. Check Minikube Status:

   ```
   minikube status
   ```

   This tells you if Minikube is running.

2. Use Minikube's Docker:

   ```
   eval $(minikube docker-env)
   ```

   This lets you build and manage Docker containers inside Minikube.

3. Create a free account in Docker Hub

Step 1: Open the Docker Hub Website

1. Open your web browser (like Chrome, Firefox, or Edge).
2. Type **https://hub.docker.com** into the address bar and press **Enter**.

Step 2: Go to the Sign-Up Page

1. Once the Docker Hub homepage loads, look for a button that says **Sign Up** (usually at the top-right corner).
2. Click on the **Sign-Up** button.

Step 3: Fill in the Registration Form

You will see a form with fields to enter your details. Fill them out as follows:

1. **Username:**
   - Choose a unique username for your Docker account (e.g., "MyDocker123").

- The username is how others might find you on Docker Hub, so pick something simple and easy to remember.

2. **Email Address:**
    - Enter a valid email address that you use frequently (e.g., myemail@example.com).
3. **Password:**
    - Create a strong password. It should have a mix of letters, numbers, and symbols (e.g., "Docker@1234").
    - Write down or save your password somewhere safe.
4. **Optional (Verify Username Availability):**
    - Sometimes, if the username is already taken, you might see a message asking you to pick a different one. Adjust your username until it's accepted.

Step 4: Accept the Terms

1. Look for a checkbox that says something like **"I agree to the Terms and Conditions."**
2. Read the terms if you wish, then tick the checkbox to agree.

Step 5: Complete the CAPTCHA

1. A CAPTCHA might appear asking you to verify that you're not a robot.
2. Follow the instructions (e.g., selecting images or typing letters/numbers) to complete the verification.

Step 6: Click Sign Up

1. After filling in all the fields and completing the CAPTCHA, click the **Sign Up** button.

Step 7: Verify Your Email Address

1. Open your email inbox (the one you used for registration).
2. Look for an email from **Docker Hub** with a subject like **"Verify Your Email."**
3. Open the email and click on the verification link inside.
    - If you don't see the email, check your **Spam** or **Promotions** folder.

Step 8: Log In to Docker Hub

1. After verifying your email, return to the Docker Hub website.
2. Click on **Log In** (usually at the top-right corner).
3. Enter your username (or email) and password.
4. Click **Log In** to access your account.

SOFTWARE ENGINEERING

UNIT - 4

TOPIC – 4

CONTAINERIZATION USING DOCKER AND DOCKER COMPOSE

Definition of Containerization:

Containerization is a way to package software and everything it needs (like code, libraries, and settings) into a single "container" so it can run anywhere—on your computer, in the cloud, or on a server—without any problems.

Description:

Imagine you have a lunchbox that keeps your food separate (rice, curry, snacks) and ready to eat anywhere. A container in DevOps works like this lunchbox. It packs your application and all its "ingredients" (dependencies) into one box, making sure it works the same way, no matter where you open it (run it).

Before containerization, moving software between environments (like from your laptop to a server) often caused issues because the environments were different. Containers solve this by providing a consistent environment inside the "box."

Example:

You create an app that needs a specific version of Python and some libraries. Normally, if you run it on a friend's computer that doesn’t have the same setup, it won’t work. But if you put your app in a container, the container includes Python, libraries, and everything else your app needs. Now, it will work perfectly on your friend’s computer, your laptop, or even on a big server.

Types of Containerization Tools:

1. Docker - Most popular tool for creating and managing containers.
2. Kubernetes - Manages and orchestrates containers at scale.
3. Podman - Similar to Docker but daemonless.
4. LXC (Linux Containers) - Lightweight container system.
5. OpenShift - Built on Kubernetes, for enterprise use.
6. Containerd - A core container runtime tool.
7. Rkt (Rocket) - Simple alternative to Docker.

Docker is popular because it makes creating, sharing, and running containers super easy!

Difference Between Jenkins and Docker

Jenkins and Docker are both tools used in software development, but they do very different jobs. Jenkins is a tool that helps automate repetitive tasks in software development. For example, when developers write code, it needs to be built (converted into a working program), tested (to make sure there are no bugs), and deployed (put into use). Instead of doing these steps manually, Jenkins can handle them automatically, saving time and reducing mistakes.

Docker is more focused on making sure your application runs smoothly anywhere. It does this by packing the application and everything it needs—like files, libraries, and settings—into something called a **container**. A container ensures that your application behaves the same way no matter where you run it, whether it's on a developer's laptop, a testing server, or in live production.

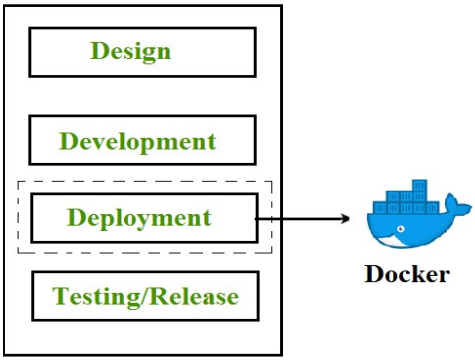
In short, Jenkins automates the process of preparing and delivering your application, while Docker ensures that the application works perfectly everywhere it is used.

Feature	Jenkins	Docker
Purpose	Automates tasks like building, testing, and deploying code.	Packages applications into containers so they run the same everywhere.

Feature	Jenkins	Docker
Focus	Streamlines the process of software development.	Ensures the app works perfectly on any computer or server.
Usage	Used for Continuous Integration/Continuous Deployment (CI/CD).	Used for creating, running, and managing containers.
What It Solves	Reduces manual work during development.	Removes compatibility issues when moving apps between computers.

Introduction to Docker

Docker is a tool that makes it easy to develop, package, and run applications. Imagine you've created an app on your laptop, and it works fine there. But when you try to run it on another computer or server, it fails because the environment is different. For example, your laptop might have a newer version of a library or software that the app needs, while the other computer has an older version.



Docker solves this problem by putting the app and all its dependencies into a **container**. This container includes everything the app needs to run, such as libraries, tools, and system settings. Once the app is in a container, it will run exactly the same way anywhere.

This is why Docker is so important in modern software development. It removes the headache of compatibility issues and makes deploying apps much easier and faster.

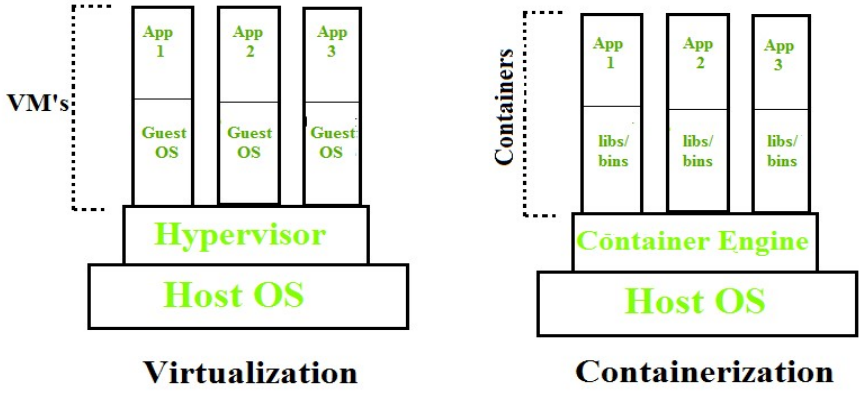
Docker and Virtual Machines

Docker is often compared to Virtual Machines (VMs) because both are used to run applications in isolated environments. But they work in very different ways.

A Docker container is a small and lightweight unit. It shares the operating system of the computer it's running on, which makes it fast and efficient. A VM, on the other hand, is heavier because it creates a separate operating system for each application.

Let's say you have three applications to run:

- With Docker, each application runs in its own container but uses the same operating system as the computer.
- With VMs, each application needs its own virtual operating system, which takes up a lot of resources.



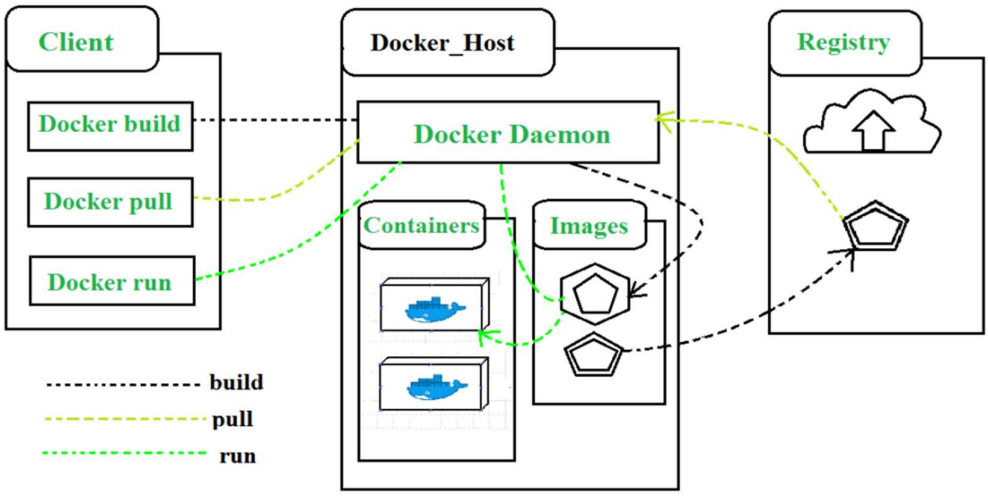
Because of this, Docker containers are faster to start, use less memory, and are easier to manage than VMs. This is why developers often prefer Docker for modern applications.

Feature	Docker (Containers)	Virtual Machines (VMs)
Size	Small and lightweight.	Heavy because each VM has its own full operating system.
Startup Speed	Very fast because it uses the host's operating system.	Slower because each VM needs to boot its own operating system.

Feature	Docker (Containers)	Virtual Machines (VMs)
Resource Use	Uses fewer resources like memory and CPU.	Uses more resources as every VM runs its own OS.
Isolation	Provides isolation but shares the host's OS.	Full isolation as each VM has its own OS.
Use Case	Good for running many lightweight applications quickly.	Useful for running applications that need very strong isolation.

Docker Architecture

Docker's architecture is how all the pieces of Docker work together. Each part has a specific role, and together they make Docker run smoothly.



1. **Docker Daemon**: This is the part of Docker that does all the heavy lifting. It runs in the background and takes care of creating, running, and managing containers.
2. **Docker Client**: This is the tool you use to interact with Docker. For example, when you type commands like `docker run`, the client sends these instructions to the Docker Daemon to execute.
3. **Docker Images**: These are like templates or blueprints. They contain everything needed to create a container, such as the application code and its dependencies.

4. **Docker Containers**: These are the actual running versions of Docker images. When you start a container, it's like bringing an image to life.
5. **Docker Registry**: This is a storage area for Docker images. Docker Hub is the most popular registry where you can find and share images.

Imagine the architecture like this:

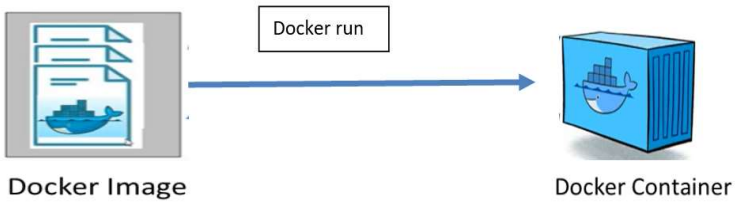
- You give a command to the **Docker Client**, which sends it to the **Docker Daemon**.
- The daemon pulls an image from the **Docker Registry** or uses a local one.
- It then uses this image to create and manage **containers** where your application runs.

This architecture makes Docker very efficient and easy to use for developers.

Key Components of Docker

To use Docker, you need to understand its main tools and features. These include:

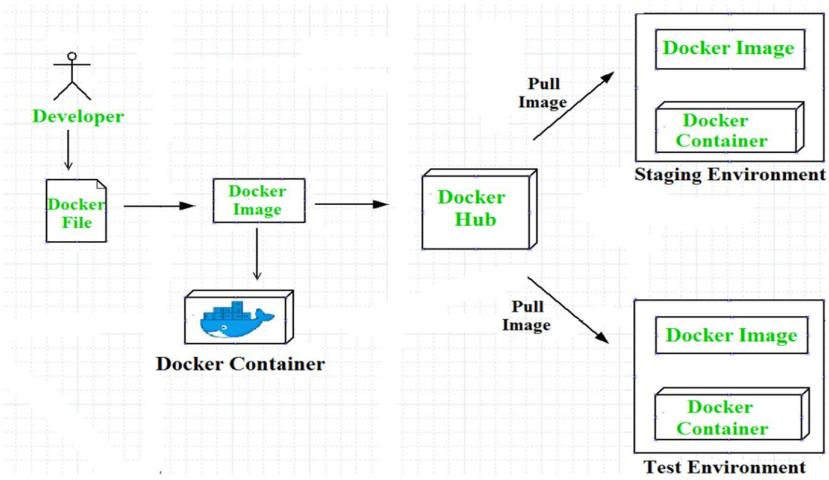
- **Docker Engine**: The main software that powers Docker. It takes care of creating and running containers.
- **Docker Image**: This is a package that has all the files and settings an application needs to run.
- **Container**: A live instance of an image where your application is running.
- **Dockerfile**: A script that lists all the steps to create a Docker image.
- **Docker Compose**: A tool that lets you manage multiple containers at the same time, making it easier to run complex applications.



How Docker Works

Using Docker typically involves three steps:

1. Create a **Dockerfile**. This file contains all the instructions Docker needs to create an image.
2. Build the image using the `docker build` command.
3. Run the image as a container using the `docker run` command.



For example, let's say you've built a web application. You would create a Dockerfile that includes the web server, your application code, and any libraries it needs. When you build the image and run it as a container, your application will be ready to use on any computer or server.

Writing a Dockerfile

A Dockerfile is a simple text file with instructions for Docker. Here's an example of a Dockerfile and what each line does:

```
FROM python:3.9
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
EXPOSE 5000
CMD ["python", "app.py"]
```

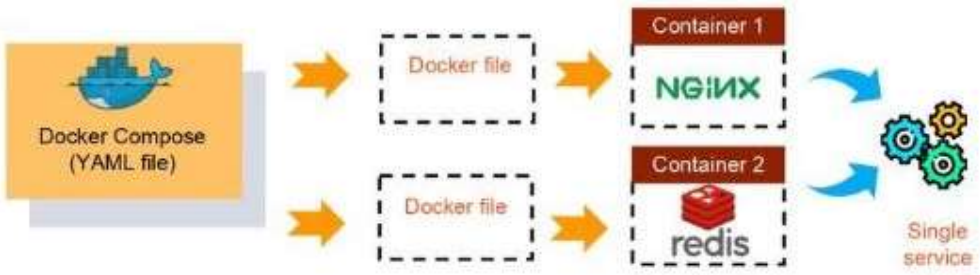
- **`FROM python:3.9`:** This sets the base image. It tells Docker to start with Python version 3.9.
- **`WORKDIR /app`:** This creates a folder called `/app` inside the container where all actions will happen.

- **`COPY . /app:`** This copies all files from your computer into the `/app` folder in the container.
- **`RUN pip install -r requirements.txt:`** This installs the Python libraries your application needs.
- **`EXPOSE 5000:`** This opens port 5000 so the application can communicate with the outside world.
- **`CMD ["python", "app.py"]:`** This starts the application by running the file `app.py`.

This Dockerfile ensures that your application has everything it needs to run properly.

Docker Compose

When your application needs more than one container, managing them manually can be difficult. For example, a web application might need one container for the web server and another for the database.



Docker Compose makes this process easier. It lets you define all the containers in a single file called **`docker-compose.yml`**. With one command, you can start or stop all the containers at once.

Here's an example **`docker-compose.yml`** file for running WordPress with a MySQL database:

```
version: '3'
services:
  wordpress:
    image: wordpress:latest
    ports:
```

```
      - "8080:80"
    environment:
      WORDPRESS_DB_HOST: db
      WORDPRESS_DB_PASSWORD: example
  db:
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: example
```

This file defines two services:

- **WordPress**: This container runs the WordPress application, accessible on port 8080.
- **Database (db)**: This container runs a MySQL database that WordPress uses to store its data.

To start both containers, you just run:

```
docker-compose up
```

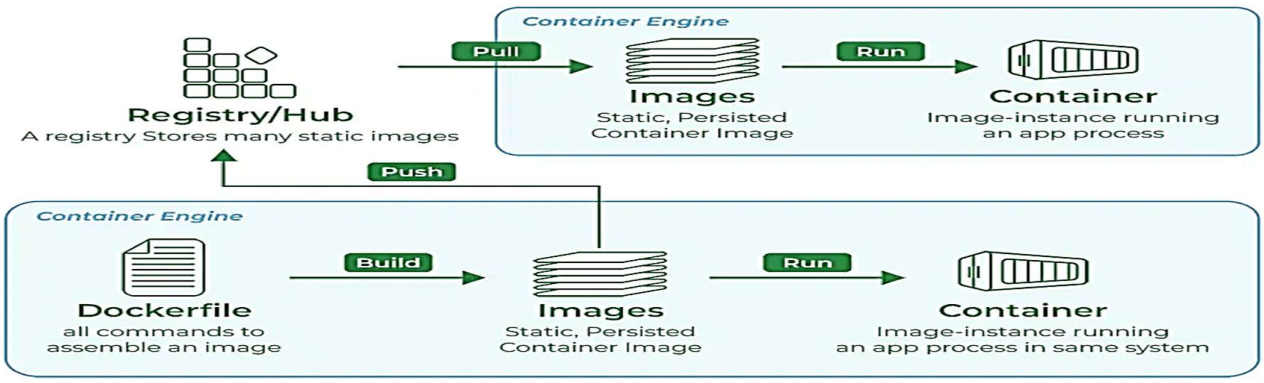
After running this command, your WordPress site will be available at `http://localhost:8080.`

Docker Hub

Docker Hub is an online library where developers can share and download Docker images. For example, if you need a basic setup like Ubuntu or MySQL, you can pull it from Docker Hub using:

```
docker pull ubuntu
```

You can also upload your own images to Docker Hub. This is useful for sharing your work with others or reusing setups in different projects.


Container Engine
Pull
Run
Registry/Hub
A registry Stores many static images
Images
Static, Persisted
Container Image
Container
Image-instance running
an app process
Push
Container Engine
Build
Run
Dockerfile
all commands to
assemble an image
Images
Static, Persisted
Container Image
Container
Image-instance running
an app process in same system

SOFTWARE ENGINEERING

UNIT - 4

TOPIC – 5

INTRODUCTION TO ORCHESTRATION USING KUBERNETES

What is Orchestration?

Definition: Orchestration refers to organizing and automating tasks to ensure they happen in the correct order and work smoothly together.

Purpose: Modern applications consist of many components like servers, databases, and containers. Manually managing these components can be time-consuming, error-prone, and complex. Orchestration simplifies this by automating these processes, saving time, and ensuring efficiency.

Example of Orchestration: Think of a wedding planner. The planner ensures that the caterer, decorator, and musicians do their tasks at the right time, so the wedding goes smoothly. Similarly, orchestration ensures all parts of a software system work together properly.

What is Container Orchestration?

Definition: Container orchestration manages containers to ensure they run properly, scale as needed, and communicate with each other efficiently.

Purpose: Containers are lightweight packages containing everything an application needs to run. While managing a single container is straightforward, handling hundreds or thousands is a daunting task. Container orchestration automates this, making it manageable.

What Does Container Orchestration Do?

- Starts containers when needed.
- Stops containers when they are no longer required.

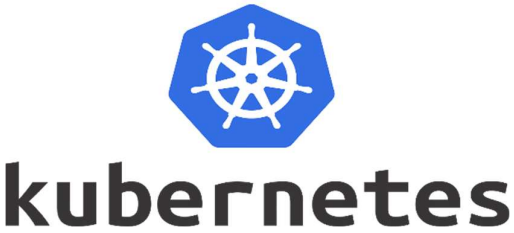
- Scales the number of containers up or down based on the workload.
- Ensures containers communicate with one another.
- Replaces failed containers.

Example of Container Orchestration: Imagine delivering lunchboxes to students. Each lunchbox contains everything needed for a meal. Managing and delivering a few lunchboxes is easy, but organizing and delivering lunchboxes to hundreds of students is challenging. Container orchestration is like a delivery system that automates this process for efficiency.

What is Kubernetes?

Definition: "Kubernetes is an open-source platform that helps automate the deployment, scaling, and management of applications inside containers."

The short form **K8s** is used for Kubernetes. The "8" refers to the eight letters between "K" and "s" in the word "Kubernetes."



Purpose: Without Kubernetes, developers must manually start, stop, and manage containers, which is time-consuming and prone to errors. Kubernetes automates these tasks, ensuring applications run reliably and scale efficiently.

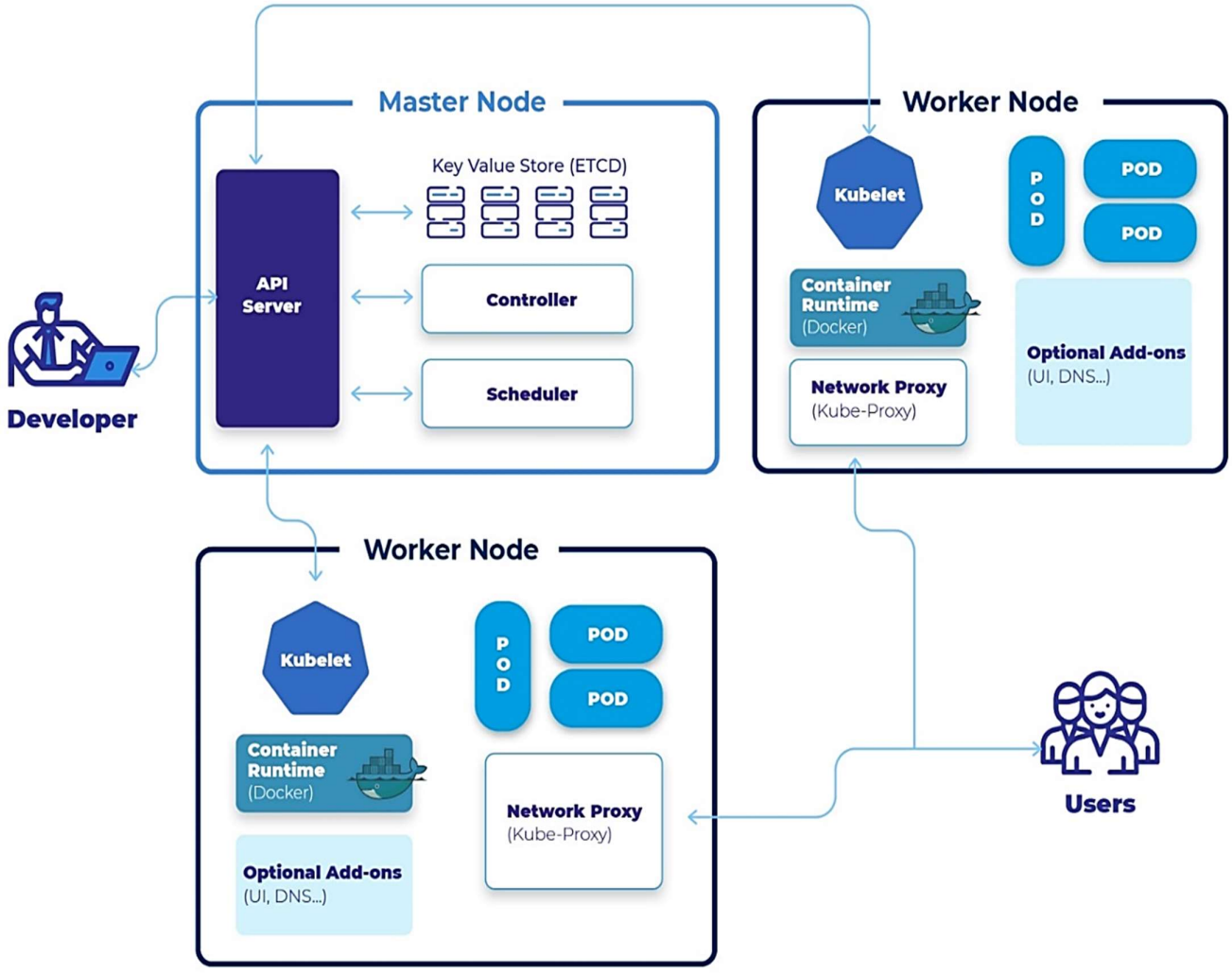
Features of Kubernetes:

- **Automatic Deployment:** Starts and stops containers as needed.
- **Scaling:** Automatically increases or decreases the number of containers based on demand.
- **Self-Healing:** Restarts containers that fail.
- **Load Balancing:** Distributes workloads evenly across containers.
- **Rolling Updates:** Updates applications without downtime.

Example of Kubernetes: Imagine running a pizza delivery service. Kubernetes acts like a manager who:

- Assigns tasks to delivery drivers (containers).
- Sends more drivers during busy hours (scaling).
- Replaces drivers who are unavailable (self-healing).

Kubernetes Architecture: Kubernetes operates like a well-organized company, where each component has a specific role:



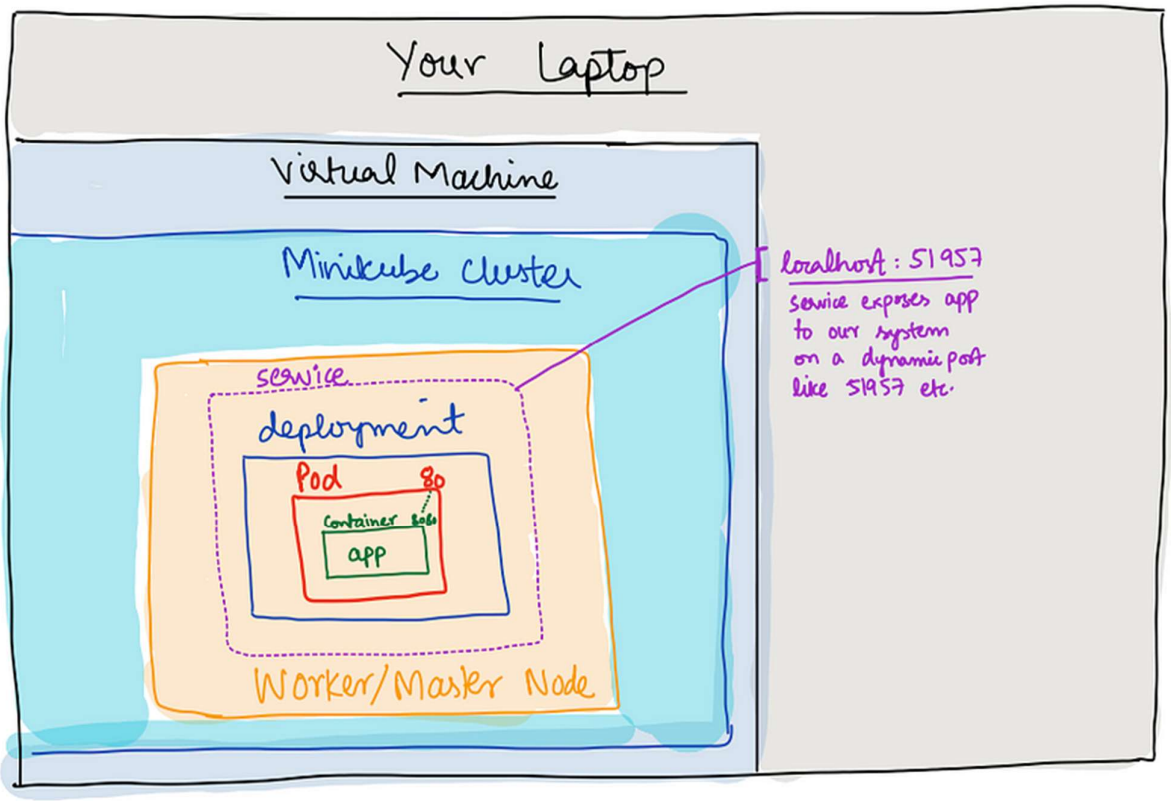
1. **Master Node:** The Master Node acts like the boss. It decides where and how containers should run, tracks the system's state, and fixes issues like restarting failed containers.
    - **API Server:** Acts as the receptionist, receiving and handling requests.

- **Scheduler:** Assigns tasks to worker nodes.
- **Controller Manager:** Ensures the system stays in the desired state, such as maintaining the correct number of containers.
- **etcd:** A database that stores information about the entire Kubernetes cluster.

2. **Worker Nodes:** Worker Nodes perform the actual work of running applications. Each worker node includes:
   - **Pods:** Groups of containers working together.
   - **Kubelet:** A helper that ensures containers run correctly.
   - **Kube Proxy:** Manages network connections between containers and external systems.
3. **Pods:** A Pod is the smallest unit in Kubernetes. It:
   - Holds one or more containers sharing resources like storage and networking.
   - Acts as a wrapper around containers to help them function together.
4. **Controller:** The Controller acts like a supervisor. It:
   - Ensures the correct number of pods are running.
   - Creates new pods if existing ones fail.
5. **Service:** A Service acts like a directory, helping users or other containers find the right pod, even if the pod moves to a different worker node.

Example of Kubernetes Architecture:

- The Master Node is like the central office managing tasks.
- Worker Nodes are like delivery vans carrying packages (pods).
- Pods are like the packages themselves.
- The Controller ensures enough vans are on the road.
- The Service connects customers to the right delivery van.



Explanation of the Diagram: This diagram represents how Kubernetes works on a local setup using Minikube, running on your laptop. Let's break it down step by step:

1. **Your laptop**: The physical machine (your laptop) is where everything is being set up.
2. **Virtual Machine**: Inside your laptop, Minikube creates a virtual machine. This is like a small computer inside your computer, and it runs the Kubernetes cluster.
3. **Minikube Cluster**: The Kubernetes cluster created by Minikube runs inside this virtual machine. The cluster includes the worker and master nodes needed for Kubernetes to function.
4. **Worker/Master Node**: Inside the Kubernetes cluster, there is at least one node (which can act as both a master and worker node in Minikube). This node handles the orchestration tasks and runs the actual containers.
5. **Pod**:

   A pod is the smallest unit in Kubernetes. It contains the actual application (or multiple applications) running in containers. In this diagram:

   - The pod includes a **container** that runs the actual application (`app`).
   - The container uses port **80** to communicate internally.

6. **Deployment**:

   The deployment is a Kubernetes object that ensures the desired number of pods are running at all times. It automatically replaces failed pods and manages updates.

7. **Service**:

   The service exposes the application (running in the pod) to the outside world. It assigns a dynamic port (like `51957`) on the local machine to make the application accessible. For example, if the application is a website, you can open a browser and go to `http://localhost:51957` to see it.

Differences Between Kubernetes, Docker, and Docker Swarm:

Kubernetes vs. Docker:

- Kubernetes and Docker work together but have different roles.
  - Docker is for creating and running containers.
  - Kubernetes organizes and manages these containers.
- **Key Differences:**

Feature	Docker	Kubernetes
Purpose	Builds and runs containers.	Manages multiple containers.
Scaling	Manual.	Automatic based on demand.
Fault Handling	Limited support.	Restarts/replaces failed containers.
Load Balancing	Basic traffic handling.	Advanced traffic management.

Example: Docker is like packing items into boxes, while Kubernetes is like a warehouse system organizing and tracking the boxes.

Kubernetes vs. Docker Swarm:

- Kubernetes is better for large, complex setups, while Docker Swarm is simpler and ideal for small projects.

- **Key Differences:**

Feature	Kubernetes	Docker Swarm
Complexity	Handles large systems.	Simpler for small setups.
Scaling	Ideal for heavy workloads.	Best for lightweight workloads.
Features	Advanced (rolling updates).	Basic features.

Example: Kubernetes is like managing a big orchestra, while Docker Swarm is like managing a small band.

What is Minikube?

Definition: Minikube is a tool that creates a small, local Kubernetes cluster on a personal computer.

Purpose: Minikube allows developers to practice Kubernetes concepts and test applications without needing a large, complex setup.

Example of Minikube: Minikube is like a small practice field where developers can try Kubernetes ideas before working on larger projects.

Deploying Applications Using Kubernetes and Minikube:

Step 1: Start Minikube Start Minikube with the command:

```
minikube start
```

This sets up a small Kubernetes cluster on the computer.

Step 2: Write a Small YAML File Create a file named `simple-app.yaml` with the following content:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
  - name: my-container
    image: nginx:latest
    ports:
    - containerPort: 80
```

What the YAML file does:

- **apiVersion:** Specifies the Kubernetes API version.
- **kind:** Defines the resource type (a Pod here).
- **metadata:** Names the pod (`my-app`).
- **spec:** Describes the container's details.
  - **name:** Names the container (`my-container`).
  - **image:** Uses the Nginx web server image.
  - **ports:** Exposes the container on port 80.

Step 3: Apply the YAML File Run the command to create the Pod:

```
kubectl apply -f simple-app.yaml
```

This instructs Kubernetes to create the pod as defined in the YAML file.

Step 4: Verify the Pod Check the running pod with:

```
kubectl get pods
```

Step 5: Clean Up Delete the pod after testing:

```
kubectl delete pod my-app
```

SOFTWARE ENGINEERING

UNIT - 4

TOPIC – 6

CONTINUOUS MONITORING USING NAGIOS

What is Continuous Monitoring?

Continuous Monitoring means regularly checking how software and systems are working to ensure everything runs smoothly. It's like having a watchful eye on your software at all times, from when it is being built to when people use it. Here's what it involves:

1. **Always Watching**: Keep track of software performance and user interactions.
2. **Automatic Checks**: Use tools to monitor and test without manual work.
3. **Instant Feedback**: Alert teams about problems quickly so they can fix them fast.
4. **Security Checks**: Look for security issues and fix them promptly.
5. **Performance Tracking**: Monitor speed, memory usage, and responsiveness.
6. **User Insights**: Understand how users are interacting with the software.
7. **Rule Checking**: Ensure the software follows required laws and standards.

It helps in identifying issues early and keeping the software efficient and user-friendly.

Why is Continuous Monitoring Important?

Continuous monitoring is important for these reasons:

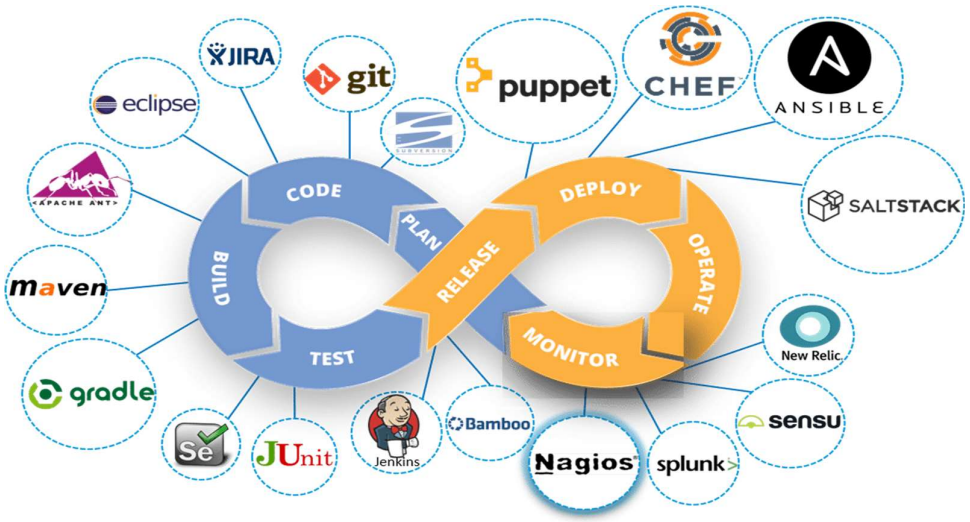
1. **Catch Problems Early**:
    - Spot and fix small issues before they turn into major problems.
    - Example: Fixing a bug before it crashes the system.
2. **Quickly Adapt to Changes**:
    - Detect and address errors caused by updates or changes.
    - Example: Finding out immediately if a new feature breaks something.

3. **Improve Over Time**:
    - Use data to enhance performance and user experience.
    - Example: Optimizing features based on user feedback.

Think of it as a system that keeps your software healthy and growing.

Tools for Continuous Monitoring

Different tools help with various aspects of continuous monitoring. Here are some examples:



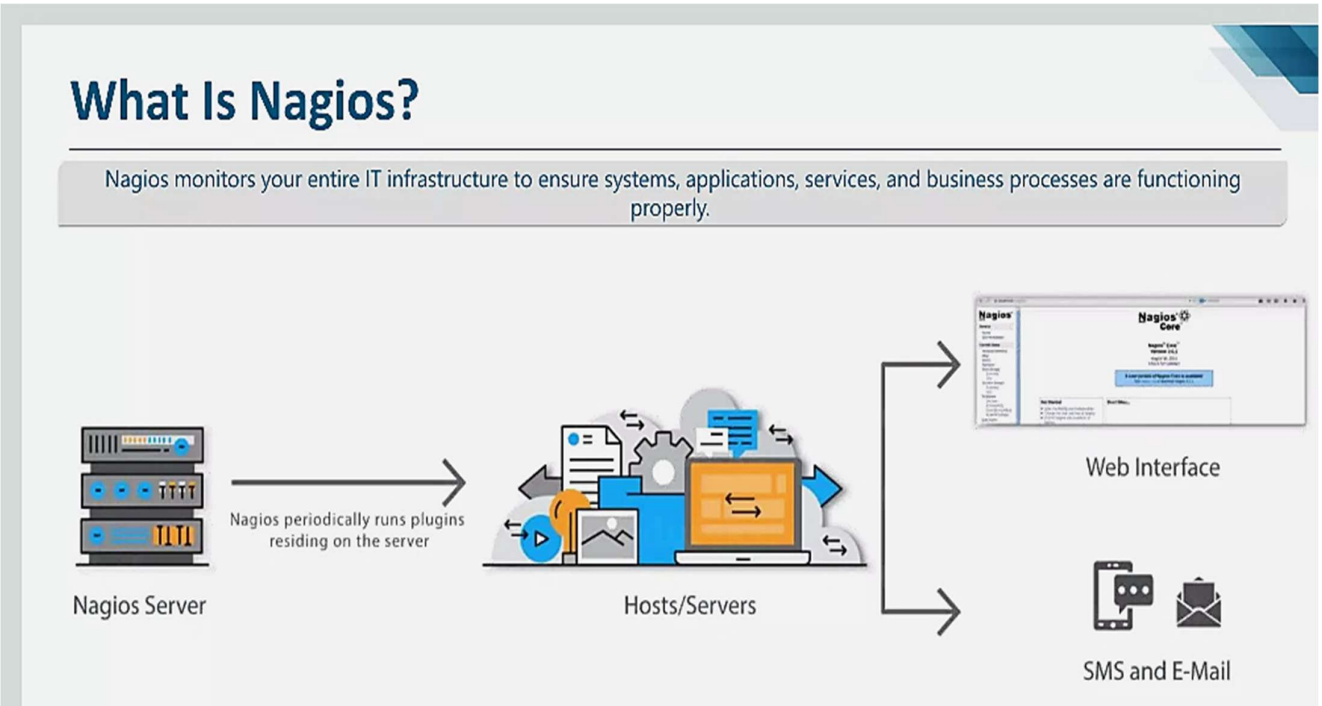
1. **Log Management Tools**:
    - Analyze system logs to find errors.
    - Example: ELK Stack (Elasticsearch, Logstash, Kibana).
2. **Performance Monitoring Tools**:
    - Check if the software is running efficiently.
    - Example: New Relic.
3. **Security Tools**:
    - Protect against threats and vulnerabilities.
    - Example: Splunk (SIEM tools).
4. **Infrastructure Monitoring Tools**:
    - Keep servers and networks running smoothly.
    - Example: Nagios.

5. **CI/CD Tools**:
    - Automate testing and deployment.
    - Example: Jenkins.
6. **User Monitoring Tools**:
    - Track user interactions with the software.
    - Example: UserZoom.
7. **Application Monitoring Tools**:
    - Find and fix issues inside the code.
    - Example: AppDynamics.
8. **Container Tools**:
    - Manage small software environments.
    - Example: Kubernetes.

These tools work together to keep your software secure, efficient, and easy to use.

What is Nagios?

Nagios is a tool that monitors your IT systems, including servers, networks, and applications. It alerts you if something isn't working as expected.



Types of Nagios:

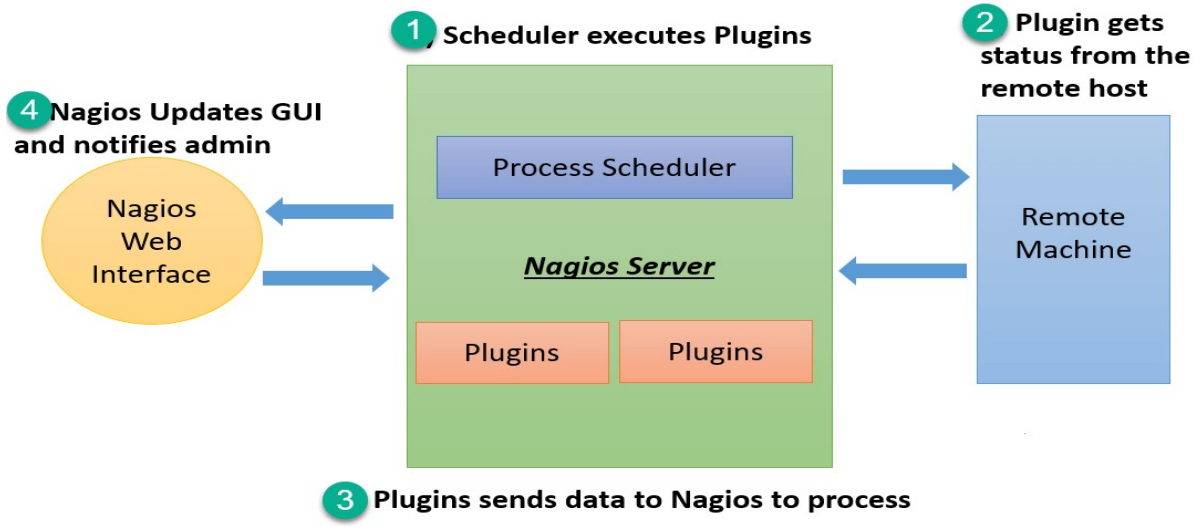
1. **Nagios Core**:
    - Free version.
    - Basic features and customizable with plugins.
2. **Nagios XI**:
    - Paid version with advanced features and technical support.

What Nagios Does:

- Checks if systems and services are running smoothly.
- Sends alerts when there are issues.
- Helps fix problems before they become serious.

Nagios Architecture

Nagios has a simple yet powerful architecture with three main parts:



1. **Nagios Core (Central Server)**:
    - Acts as the brain of the system.
    - Manages checks and collects results from plugins.
2. **Plugins (Workers)**:
    - Perform specific tasks, like checking server health or available storage.

- Send back results to the Core.

3. **Web Interface (Dashboard)**:
   - Displays monitoring data in an easy-to-read format.
   - Lets you configure settings, view alerts, and manage systems.

How It Works:

1. Nagios Core tells the plugins what to check and when.
2. Plugins collect data and send it back to the Core.
3. The Core processes the data and decides if there's a problem.
4. Alerts are displayed on the web interface or sent via email/messages.

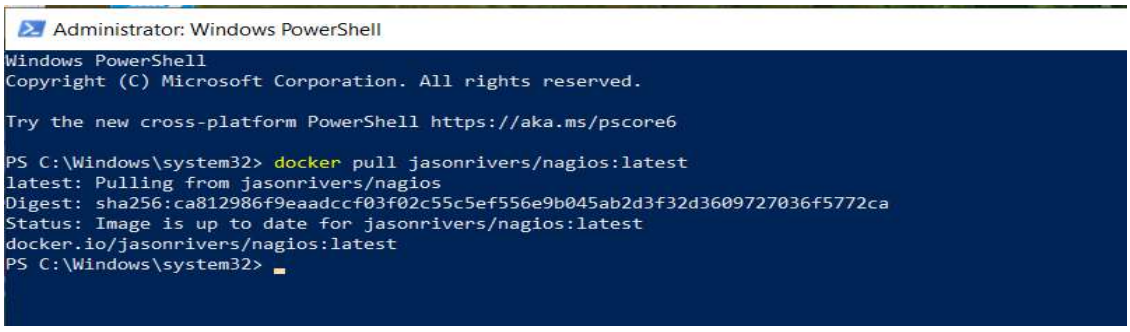
This setup ensures you always know the health of your IT systems.

Installing Nagios Using Docker

You can easily set up Nagios using Docker. Here's how:

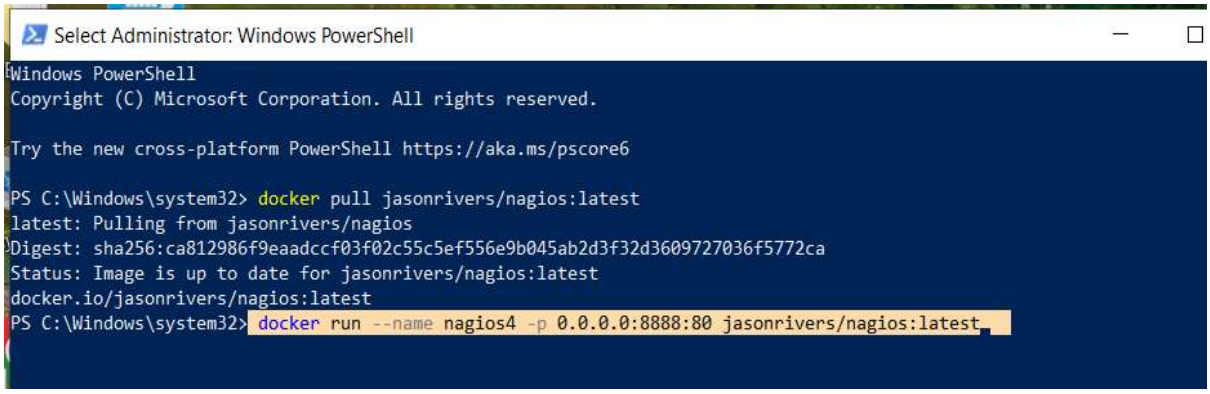
1. **Prepare Docker**:
   - Install Docker Desktop and make sure it is running.
2. **Download Nagios Image**:
   - In the terminal, type:

```
docker pull jasonrivers/nagios:latest
```



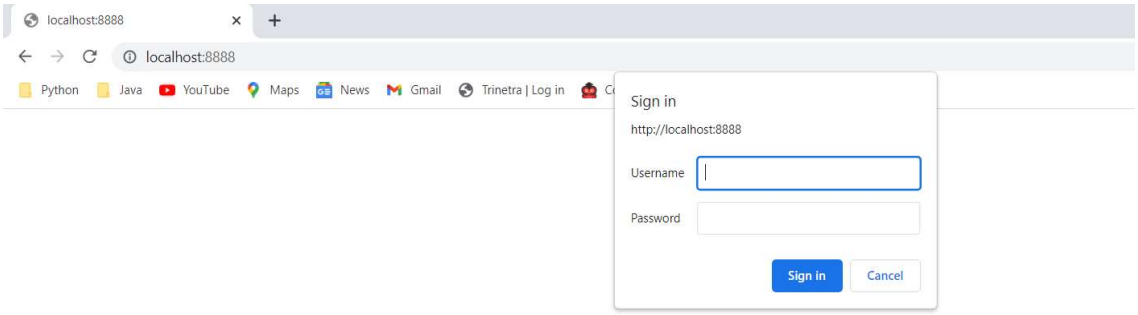
3. **Run the Nagios Container**:
   - Type the following command:

`docker run -d --name nagiosdemo -p 8888:80 jasonrivers/nagios:latest`



- This sets up Nagios on your computer.

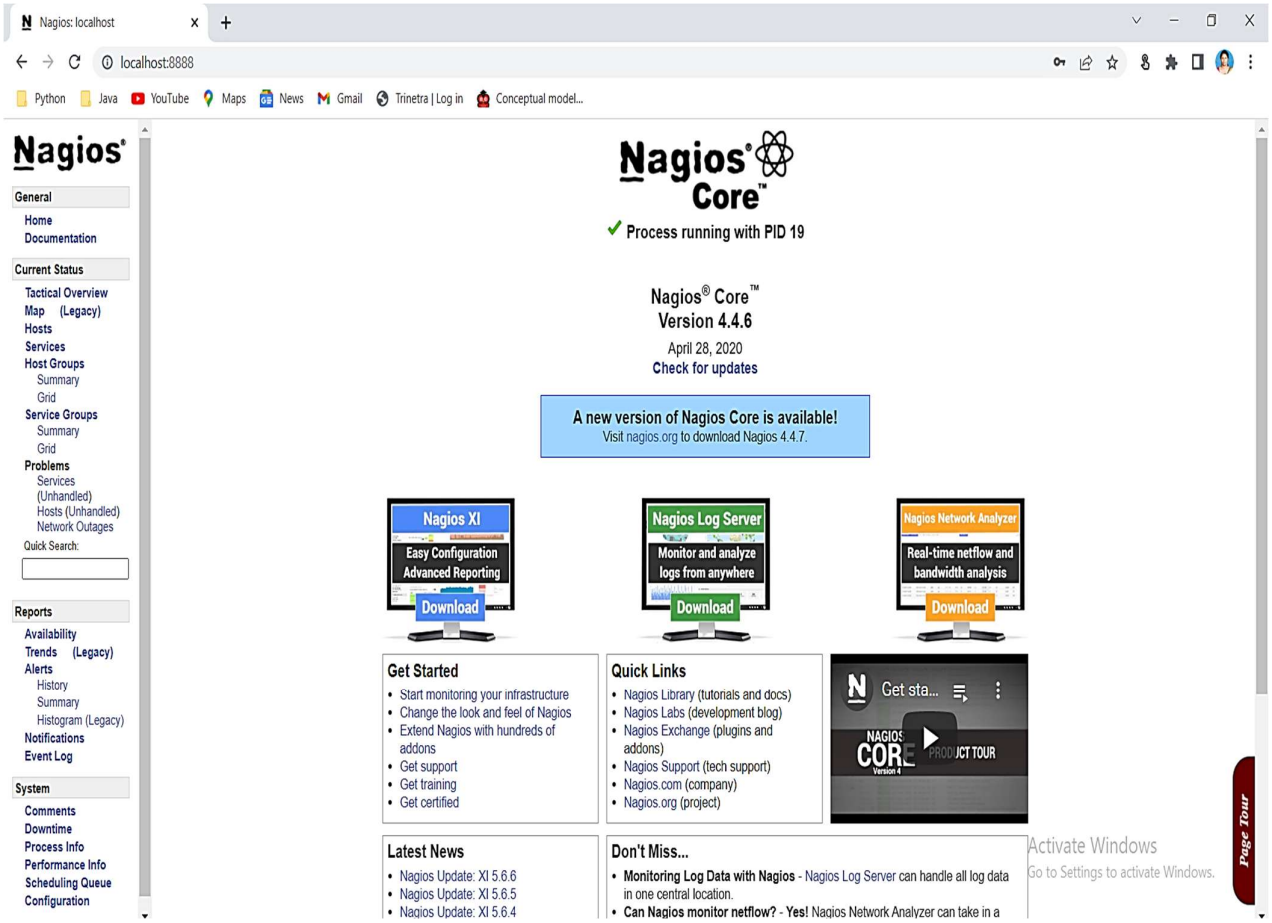
4. **Access Nagios**:
   - Open your browser and go to: http://localhost:8888/nagios.



   - Login credentials:
     - Username: `nagiosadmin`
     - Password: `nagios`
5. **Stop or Remove Nagios**:
   - To stop: `docker stop nagiosdemo`
   - To remove: `docker rm nagiosdemo`

Nagios Dashboard

The Nagios dashboard is your control panel for monitoring. It includes:



1. **Host Status**:
   - Shows if servers and devices are working fine.
2. **Service Status**:
   - Displays the status of services like email or websites.
3. **Summary**:
   - Gives an overview of all monitored systems.
4. **Performance Data**:
   - Shows metrics like speed and resource usage.
5. **Notifications**:
   - Lists alerts for any problems.
6. **Event Log**:
   - Keeps track of past events and changes.
7. **Configuration Options**:
   - Lets you customize what to monitor.

With this dashboard, you can quickly identify and resolve issues.

Why Use Nagios for Continuous Monitoring?

Nagios helps by:

- Detecting problems early.
- Sending alerts for quick fixes.
- Keeping your systems running efficiently.

It’s like having a reliable assistant that never stops watching over your IT setup.