­­­­­­Docker

Steps to Install Docker :

Open your terminal (or command prompt) on your local machine.

Connect to your EC2 instance using SSH. Replace your-key-pair.pem, your-ec2-user, and your-ec2-public-dns with your key pair file, EC2 username (typically ubuntu), and the public DNS of your EC2 instance, respectively.

Cmd:

ssh -i your-key-pair.pem your-ec2-user@your-ec2-public-dns

Step 2: Update the Package Index

Update your package index and install required dependencies:

Cmd:

sudo apt-get update

Step 3: Install Docker

Install Docker's package dependencies:

Cmd:

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

Package List

apt-transport-https:

Purpose: A package that allows the use of HTTPS in sources.list for apt-get.

Function: Ensures that apt-get can retrieve packages from repositories over secure HTTPS connections, providing an added layer of security.

ca-certificates:

Purpose: A package that installs a set of Certificate Authority (CA) certificates.

Function: Ensures that the system can validate SSL/TLS certificates, which is necessary for establishing secure connections over HTTPS.

curl:

Purpose: A command-line tool for transferring data with URLs.

Function: Often used to download content from the internet, making HTTP requests, and working with various protocols.

software-properties-common:

Purpose: A package that provides a set of utility scripts for managing software repositories.

Function: Includes commands like add-apt-repository which simplifies the process of adding new software repositories and PPAs (Personal Package Archives).

Add Docker’s official GPG key:

Cmd:

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add –

GPG, or GNU Privacy Guard, is a cryptographic software tool that enables secure communication and data storage. It is widely used for encrypting data and creating digital signatures.

Purpose of the Docker GPG Key

Authentication: The GPG key verifies that the Docker packages you download are indeed from the official Docker repository. This helps prevent man-in-the-middle attacks where an attacker might try to serve malicious packages.

Integrity: The GPG key ensures that the Docker packages have not been tampered with. When a package is signed with a GPG key, any alteration to the package will be detected because the signature will no longer match.

Cmd:

sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb\_release -cs) stable"

The command you provided adds Docker's official repository to your system's list of repositories.

Update the package index again:

Cmd:

sudo apt-get update

Install Docker:

Cmd:

sudo apt-get install -y docker-ce

Verify Docker installation:

Cmd:

sudo docker –version

sudo docker -v

Step 4: Manage Docker as a Non-root User (Optional)

Create the docker group:

Cmd:

sudo groupadd docker

Add your user to the docker group:

Cmd:

sudo usermod -aG docker $USER

Log out and log back in to apply the group membership.

Step 5: Start Docker

Cmd:

sudo systemctl start docker

sudo systemctl enable docker

Get docker status know

Cmd:

sudo systemctl status docker

Step 6: Install NGINX in a Docker Container

Pull the NGINX Docker image:

Cmd:

sudo docker pull nginx

Run an NGINX container:

Cmd:

sudo docker run --name my-nginx -d -p 80:80 nginx

The default document root for NGINX inside a Docker container is typically located at /usr/share/nginx/html. This is where NGINX serves static files from by default.

Steps to Access the NGINX Document Root

Enter the Running NGINX Container: First, you need to get inside the running NGINX container.

Cmd:

sudo docker exec -it my-nginx /bin/bash

Navigate to the Document Root: Once inside the container, navigate to the document root.

Cmd:

cd /usr/share/nginx/html

--transfer file to docker image

Cmd:

cd mywebsite/

nano index.html

sudo docker cp index.html my-nginx:/usr/share/nginx/html/index.html

Copying a Directory:

Cmd:

sudo docker cp /path/to/local/directory/. my-nginx:/usr/share/nginx/html/

Example:

Local pc:

Cmd:

ssh -i "InstanceKey.pem" ubuntu@ec2-16-16-210-217.eu-north-1.compute.amazonaws.com

scp -i "c:/Users/jains/Downloads/InstanceKey.pem" -r "e:/webtech6pm/" [ubuntu@ec2-16-16-210-217.eu-north-1.compute.amazonaws.com:/home/ubuntu/mywebsite](mailto:ubuntu@ec2-16-16-210-217.eu-north-1.compute.amazonaws.com:/home/ubuntu/mywebsite)

ec2 pc:

Cmd:

sudo docker cp /home/ubuntu/mywebsite/webtech6pm/. my-nginx:/usr/share/nginx/html/

Example Dockerfile with Custom HTML

If you want to create a Docker image with a custom HTML file, you can use a Dockerfile:

Create a Dockerfile:

Cmd:

FROM nginx:latest

COPY ./my-website /usr/share/nginx/html

In this Dockerfile:

FROM nginx:latest uses the official NGINX image.

COPY ./my-website /usr/share/nginx/html copies the content of your local my-website directory to the document root in the container.

Build the Docker Image:

Cmd:

sudo docker build -t my-custom-nginx .

sudo docker run --name my-nginx -d -p 80:80 my-custom-nginx

list of images:

Cmd:

docker ps // active images

docker ps -a // active, stop both images

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

d9b100f2f636 nginx "nginx -g 'daemon of…" 3 minutes ago

stop image using container id

Cmd:

docker stop d9b100f2f636

………………………………………………………………………………………………………………………………….

A CI/CD pipeline is a series of steps that automate the processes of Continuous Integration (CI) and Continuous Delivery (CD) or Continuous Deployment (CD). It is a fundamental DevOps practice aimed at improving software development and delivery by ensuring that code changes are automatically tested, integrated, and deployed to production or staging environments. Here's a detailed look at what CI/CD pipelines are and how they work:

### Continuous Integration (CI)

\*\*Continuous Integration (CI)\*\* is the practice of merging all developers’ working copies to a shared mainline several times a day. The primary goals of CI are to detect integration issues early and improve the quality of the software.

#### Key Aspects of CI:

1. \*\*Source Code Integration\*\*: Developers frequently commit code changes to a shared repository.

2. \*\*Automated Builds\*\*: Every commit triggers an automated build process, compiling the code and generating binaries if necessary.

3. \*\*Automated Testing\*\*: Automated tests (unit tests, integration tests, etc.) are run against the new code to verify its correctness and stability.

### Continuous Delivery (CD)

\*\*Continuous Delivery (CD)\*\* is the practice of ensuring that the codebase is always in a deployable state. The goal is to make deployment a routine activity that can be done on demand.

#### Key Aspects of CD:

1. \*\*Deployment Automation\*\*: Once the code passes all tests, it is automatically prepared for deployment to a staging environment.

2. \*\*Manual Approval\*\*: Deployment to production often requires manual approval, ensuring that changes are reviewed before going live.

3. \*\*Release Readiness\*\*: The application is always in a state where it can be released to production.

### Continuous Deployment (CD)

\*\*Continuous Deployment (CD)\*\* takes continuous delivery a step further by automatically deploying every change that passes the automated tests to production without manual intervention.

#### Key Aspects of Continuous Deployment:

1. \*\*Fully Automated Pipeline\*\*: The entire process from code commit to deployment in production is automated.

2. \*\*Immediate Feedback\*\*: Developers get immediate feedback on their changes as they are deployed live.

3. \*\*No Manual Approval\*\*: Changes are deployed to production without any manual approval steps.

### Components of a CI/CD Pipeline

1. \*\*Version Control System (VCS)\*\*: The source code repository where developers commit their code (e.g., GitHub, GitLab, Bitbucket).

2. \*\*CI/CD Server\*\*: A server that orchestrates the CI/CD pipeline (e.g., Jenkins, CircleCI, Travis CI, GitLab CI/CD).

3. \*\*Build Tools\*\*: Tools that automate the building of the application (e.g., Maven, Gradle).

4. \*\*Automated Testing Tools\*\*: Tools that run automated tests (e.g., JUnit, Selenium).

5. \*\*Artifact Repository\*\*: A repository where built artifacts (e.g., binaries, Docker images) are stored (e.g., Nexus, Artifactory).

6. \*\*Deployment Tools\*\*: Tools that automate the deployment of the application to different environments (e.g., Ansible, Kubernetes, Terraform).

### Typical Workflow of a CI/CD Pipeline

1. \*\*Code Commit\*\*: Developers commit code changes to the version control system.

2. \*\*Build\*\*: The CI/CD server detects the commit and triggers a build process to compile the code.

3. \*\*Test\*\*: Automated tests are run to verify the code.

4. \*\*Artifact Creation\*\*: If tests pass, an artifact (e.g., a Docker image) is created and stored in an artifact repository.

5. \*\*Deploy to Staging\*\*: The artifact is deployed to a staging environment for further testing and validation.

6. \*\*Manual Approval (for CD)\*\*: For continuous delivery, a manual approval step may be required to deploy the artifact to production.

7. \*\*Deploy to Production\*\*: The artifact is deployed to the production environment.

8. \*\*Monitoring and Feedback\*\*: The deployed application is monitored, and feedback is provided to developers.

### Benefits of CI/CD

- \*\*Reduced Integration Issues\*\*: Early detection of integration problems.

- \*\*Faster Release Cycles\*\*: Automating the build, test, and deployment processes speeds up the release cycles.

- \*\*Improved Code Quality\*\*: Automated testing ensures that code changes are verified before integration.

- \*\*Consistency and Reliability\*\*: Automated processes ensure consistent and reliable deployment.

### Summary

A CI/CD pipeline is an automated sequence of steps that streamline the processes of integrating, testing, and deploying code changes. It involves continuous integration to merge code changes frequently, continuous delivery to ensure the code is always in a deployable state, and optionally continuous deployment to automatically deploy every change to production. CI/CD pipelines improve software quality, reduce integration issues, and enable faster and more reliable software delivery.

………………………………………………………………………………………………………

The evolution of cloud computing and DevOps represents a significant shift in how software development, deployment, and IT operations are managed. Both have developed in response to the need for more agile, scalable, and efficient IT solutions. Here’s a detailed overview of their evolution:

Evolution of Cloud Computing

1. \*\*Early Days: Mainframes and Client-Server Models (1950s-1990s)\*\*

- \*\*Mainframes\*\*: Centralized computing with terminals accessing a powerful central computer.

- \*\*Client-Server Architecture\*\*: Distributed computing model where client devices interact with server resources.

2. \*\*Virtualization (1990s-2000s)\*\*

- \*\*Emergence of Virtual Machines (VMs)\*\*: Enabled multiple operating systems to run on a single physical server, increasing hardware utilization and flexibility.

- \*\*VMware\*\*: Founded in 1998, popularized virtualization technology, paving the way for cloud computing.

3. \*\*The Rise of Cloud Computing (2000s-Present)\*\*

- \*\*Infrastructure as a Service (IaaS)\*\*: Provision of virtualized computing resources over the internet. Examples: Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP).

- \*\*Platform as a Service (PaaS)\*\*: Provides a platform allowing customers to develop, run, and manage applications without dealing with underlying infrastructure. Examples: Heroku, Google App Engine.

- \*\*Software as a Service (SaaS)\*\*: Software delivery model in which applications are hosted by a service provider and made available to customers over the internet. Examples: Salesforce, Google Workspace, Microsoft Office 365.

4. \*\*Cloud-Native and Serverless Computing (2010s-Present)\*\*

- \*\*Cloud-Native Applications\*\*: Designed to leverage cloud environments, utilizing microservices architecture, containers, and Kubernetes for orchestration.

- \*\*Serverless Computing\*\*: Abstracts server management entirely, allowing developers to focus on code while the cloud provider manages infrastructure. Examples: AWS Lambda, Google Cloud Functions.

Evolution of DevOps

1. \*\*Traditional IT Operations (Pre-2000s)\*\*

- \*\*Siloed Teams\*\*: Development and operations teams worked independently, often leading to inefficiencies and slow release cycles.

- \*\*Waterfall Model\*\*: Sequential development process with distinct phases, making it difficult to incorporate changes once a phase is completed.

2. \*\*Agile Development (2000s)\*\*

- \*\*Agile Manifesto (2001)\*\*: Introduced principles for more flexible, iterative, and collaborative development processes.

- \*\*Scrum and Kanban\*\*: Popular agile methodologies that emphasize iterative progress, continuous feedback, and cross-functional teams.

3. \*\*Birth of DevOps (Late 2000s)\*\*

- \*\*DevOps Movement\*\*: Coined in 2009 by Patrick Debois, DevOps emphasizes collaboration between development and operations to improve efficiency and deployment frequency.

- \*\*CAMS Model\*\*: Culture, Automation, Measurement, and Sharing – core principles of DevOps introduced by John Willis and Damon Edwards.

4. \*\*DevOps Practices and Tools (2010s-Present)\*\*

- \*\*Continuous Integration (CI)\*\*: Merging code changes frequently and automatically testing them to detect issues early.

- \*\*Continuous Delivery (CD)\*\*: Ensuring code is always in a deployable state, with automated deployment to staging environments.

- \*\*Infrastructure as Code (IaC)\*\*: Managing and provisioning computing infrastructure through machine-readable definition files. Examples: Terraform, AWS CloudFormation.

- \*\*Configuration Management\*\*: Tools to automate the deployment and configuration of infrastructure. Examples: Ansible, Chef, Puppet.

- \*\*Containerization\*\*: Encapsulating applications and their dependencies into containers for consistent deployment across environments. Examples: Docker, Kubernetes.

- \*\*Monitoring and Logging\*\*: Tools to monitor system performance and log application behavior for proactive issue resolution. Examples: Prometheus, Grafana, ELK Stack (Elasticsearch, Logstash, Kibana).

### Integration of Cloud and DevOps

1. \*\*Cloud as a Catalyst for DevOps\*\*

- \*\*Scalability and Flexibility\*\*: Cloud computing provides the scalability and flexibility required for DevOps practices, enabling rapid provisioning and de-provisioning of resources.

- \*\*Global Accessibility\*\*: Cloud services can be accessed from anywhere, facilitating collaboration among distributed teams.

2. \*\*DevOps Enhancing Cloud Adoption\*\*

- \*\*Automation\*\*: DevOps practices automate cloud infrastructure management, making it easier to scale and deploy applications in the cloud.

- \*\*Efficiency\*\*: Continuous integration and continuous delivery (CI/CD) pipelines streamline the deployment process, making the most of cloud capabilities.

### Key Trends and Future Directions

1. \*\*Hybrid and Multi-Cloud Strategies\*\*

- Organizations are increasingly adopting hybrid cloud (combining private and public clouds) and multi-cloud strategies (using multiple cloud providers) to avoid vendor lock-in and optimize costs and performance.

2. \*\*AI and Machine Learning Integration\*\*

- Cloud providers are offering advanced AI and ML services, enabling developers to build intelligent applications with ease.

3. \*\*Edge Computing\*\*

- Bringing computation and data storage closer to the sources of data (e.g., IoT devices) to reduce latency and bandwidth use.

4. \*\*Security and Compliance\*\*

- As cloud adoption grows, so does the focus on cloud security and compliance, with DevOps practices incorporating security (DevSecOps) to ensure secure software delivery.

### Summary

The evolution of cloud computing and DevOps reflects a shift towards more efficient, scalable, and agile IT practices. Cloud computing has transformed the way resources are provisioned and managed, while DevOps has revolutionized software development and operations by fostering collaboration and automation. Together, they enable organizations to innovate faster, improve quality, and respond more effectively to changing business needs.

**Introduction to Microservices**

**Microservices Architecture**

Microservices architecture is an architectural style that structures an application as a collection of small, autonomous services modeled around a business domain. Each microservice is a self-contained unit that encapsulates its own data and business logic, communicating with other services through APIs. This approach contrasts with monolithic architecture, where an application is built as a single, indivisible unit.

### Fragmentation of Business Requirements

In microservices architecture, business requirements are fragmented into smaller, manageable services. Each service is designed to handle a specific business function. This fragmentation allows teams to develop, deploy, and scale services independently, leading to increased agility and faster time-to-market. For example, in an e-commerce application, different microservices might handle user authentication, product catalog management, and order processing.

### Deployment Patterns

There are various deployment patterns for microservices, including:

1. \*\*Single Service per Host:\*\* Each microservice is deployed on a separate host, such as a VM or physical server. This provides strong isolation but can be resource-intensive.

2. \*\*Multiple Services per Host:\*\* Multiple microservices are deployed on the same host, often within containers. This improves resource utilization but requires careful management of resource contention and isolation.

3. \*\*Serverless Deployment:\*\* Microservices are deployed as functions in a serverless architecture, such as AWS Lambda. This abstracts away server management and allows automatic scaling based on demand.

### API Gateway

An API Gateway is an essential component in microservices architecture, acting as a reverse proxy to route client requests to appropriate microservices. It provides a single entry point for all client interactions, handling tasks such as authentication, rate limiting, logging, and request transformation. By offloading these concerns to the API Gateway, microservices can remain focused on their core business logic.

### Service Discovery

Service Discovery is a mechanism that enables microservices to locate each other dynamically in a distributed system. It involves two key components:

1. \*\*Service Registry:\*\* A database where microservices register their locations (e.g., IP addresses, ports) upon startup. Popular service registries include Consul, Eureka, and etcd.

2. \*\*Service Discovery Client:\*\* Embedded in microservices, this client queries the service registry to find the network locations of other services.

### Database Management for Microservices

Managing databases in a microservices architecture requires careful planning to maintain data consistency and integrity. Strategies include:

1. \*\*Database per Service:\*\* Each microservice has its own database, encapsulating its data within its boundaries. This approach ensures loose coupling and autonomy but introduces complexity in maintaining data consistency across services.

2. \*\*Shared Database:\*\* Multiple microservices share a common database schema. This simplifies data management but can lead to tight coupling and scalability issues.

3. \*\*Event Sourcing and CQRS:\*\* These patterns help manage data consistency by decoupling read and write operations. Event Sourcing involves storing all changes as a sequence of events, while CQRS (Command Query Responsibility Segregation) separates read and write operations into different models.

In conclusion, microservices architecture provides significant benefits in terms of scalability, flexibility, and resilience. However, it also introduces challenges, particularly in areas like service coordination, data management, and deployment complexity. Understanding these core concepts is crucial for successfully implementing and managing microservices.

/////////////////////////////////

**Managing databases in a microservices** architecture can be complex due to the distributed nature of the system. Here are some key strategies and best practices for database management in microservices:

### Database per Microservice

\*\*1. Database per Service:\*\*

   - Each microservice has its own private database.

   - Ensures loose coupling between services, as each service manages its own data independently.

   - Facilitates independent scaling and deployment of services.

\*\*Advantages:\*\*

   - Autonomy: Services can choose the most suitable database technology for their specific needs.

   - Isolation: Changes in one service's database schema do not affect other services.

\*\*Challenges:\*\*

   - Data Consistency: Maintaining consistency across distributed databases can be difficult.

   - Transactions: Distributed transactions (spanning multiple services) are complex and may require patterns like Saga.

### Shared Database

\*\*2. Shared Database:\*\*

   - Multiple microservices share a common database schema.

   - Simplifies data management by maintaining a single source of truth.

\*\*Advantages:\*\*

   - Simplified Data Access: Services can easily access and update data without complex coordination.

   - Easier Transactions: Transactions involving multiple services are simpler to implement.

\*\*Challenges:\*\*

   - Coupling: Tight coupling between services can hinder independent deployment and scaling.

   - Schema Changes: Changes to the shared schema must be coordinated across all services.

### Patterns for Data Management

\*\*3. Event Sourcing:\*\*

   - Instead of storing the current state, all changes (events) to the state are stored.

   - Provides a complete audit trail and allows the system to rebuild state by replaying events.

\*\*Advantages:\*\*

   - Traceability: Every change is recorded, providing a clear history of how the data evolved.

   - Scalability: Services can handle high loads by distributing event processing.

\*\*Challenges:\*\*

   - Complexity: Implementing and managing an event store requires significant effort.

   - Eventual Consistency: The system may not be immediately consistent after an event is processed.

\*\*4. CQRS (Command Query Responsibility Segregation):\*\*

   - Separates read and write operations into different models.

   - The command model handles writes, while the query model handles reads.

\*\*Advantages:\*\*

   - Optimized Read/Write Performance: Each model can be optimized for its specific purpose.

   - Scalability: Read and write workloads can be scaled independently.

\*\*Challenges:\*\*

   - Complexity: Requires managing two different models and ensuring they remain consistent.

   - Data Synchronization: Keeping the read model in sync with the write model can be challenging.

\*\*5. Saga Pattern:\*\*

   - A distributed transaction pattern that ensures data consistency across multiple services.

   - Breaks down a large transaction into smaller, individual steps (local transactions), with each step being managed by a single service.

\*\*Advantages:\*\*

   - Data Consistency: Ensures eventual consistency across services.

   - Resilience: Each step of the transaction can be independently retried or compensated.

\*\*Challenges:\*\*

   - Complexity: Implementing sagas requires careful coordination and error handling.

   - Latency: May introduce additional latency due to the asynchronous nature of the transactions.

### Best Practices

\*\*Data Ownership:\*\*

   - Clearly define which service owns which data. Each piece of data should have a single owner to avoid conflicts.

\*\*API Contracts:\*\*

   - Use well-defined API contracts for data exchange between services. This ensures clarity and stability in communication.

\*\*Data Replication:\*\*

   - Where necessary, replicate data across services to improve performance and resilience. Use eventual consistency mechanisms to handle synchronization.

\*\*Automated Testing and Monitoring:\*\*

   - Implement automated testing for database interactions to catch issues early.

   - Use monitoring tools to track the performance and health of databases.

In summary, effective database management in microservices involves balancing autonomy and data consistency, using appropriate patterns like Event Sourcing, CQRS, and Sagas, and following best practices for data ownership, API contracts, and replication.