DevOps

- **Introduction to DevOps**
- **Linux Fundamentals (Mandatory for DevOps)**
- **Version Control Systems**
- **Build Tools**
- **Continuous Integration (CI)**
- **Continuous Testing**
- **Continuous Deployment & Delivery (CD)**
- **Containerization(Docker)**
- **Container Orchestration(Kubernetes)**
- **Infrastructure as Code (IaC)**
- **Cloud Computing for DevOps**
- **Monitoring & Logging**
- **Security in DevOps (DevSecOps)**
- **Networking & System Design for DevOps**
- **Advanced DevOps Topics**

1.1 What is DevOps?

Simple Definition:
DevOps is a way of working that brings **software developers (Dev)** and **operations teams (Ops)** together so they can **build, test, and release software faster, more reliably, and with fewer mistakes**.

Think of it like a football team:

- Developers are the players who pass the ball (write code).
- Operations are the goalkeepers who make sure nothing breaks (keep systems running).
- DevOps is the coach who makes sure everyone plays together smoothly.

Key Idea:

- Breaks the wall between "I built it" and "You run it."
- Encourages **collaboration**, **automation**, and **continuous improvement**.

1.2 DevOps vs Agile vs Traditional IT

Feature	Traditional IT	Agile	DevOps
Work Style	Separate teams for Dev & Ops	Iterative development for Dev team only	Dev & Ops work together
Release Cycle	Months/Years	Weeks	Days/Hours
Automation	Very little	Mostly in development	End-to-end automation
Focus	Stability	Faster development	Fast + stable delivery
Communication	Minimal	Good within Dev team	Strong between Dev & Ops

Shortcut Memory Trick:

- **Traditional IT** = Siloed teams, slow.
- **Agile** = Fast development, Ops separate.
- **DevOps** = Fast development + smooth operations together.

1.3 DevOps Benefits & Challenges

Benefits:
Faster delivery of features
Higher software quality
Better collaboration between teams

Fewer errors in production
Continuous feedback for improvement

Challenges:
Requires cultural change (teams must work together)
Need to learn new tools & practices
Automation setup can take time initially
Resistance from people used to old ways

1.4 DevOps Lifecycle & Stages

Lifecycle Stages:

1. **Plan** – Define requirements, assign tasks (Tools: Jira, Trello)
2. **Code** – Write the application (Tools: Git, GitHub)
3. **Build** – Compile code, create packages (Tools: Maven, Gradle)
4. **Test** – Automated tests to find bugs early (Tools: Selenium, JUnit)
5. **Release** – Prepare the final version for users
6. **Deploy** – Put the software into production (Tools: Docker, Kubernetes)
7. **Operate** – Keep it running smoothly
8. **Monitor** – Watch performance, fix issues quickly (Tools: Prometheus, Grafana)

This is a continuous cycle — after monitoring, feedback goes back to planning.

1.5 Key DevOps Principles (CAMS)

1. **Culture** – Break silos, promote trust & teamwork.
2. **Automation** – Automate builds, testing, deployments.
3. **Measurement** – Track performance, deployment speed, errors.
4. **Sharing** – Share knowledge, successes, and failures openly.

Easy Memory Tip: CAMS = Culture + Automation + Measurement + Sharing.

1.6 Overview of DevOps Toolchain

Toolchain = The set of tools used at different stages of the DevOps lifecycle.

Stage	Common Tools
Plan	Jira, Trello, Confluence
Code	Git, GitHub, GitLab, Bitbucket
Build	Maven, Gradle, npm
Test	Selenium, JUnit, PyTest

Stage	Common Tools
Release	Jenkins, GitHub Actions
Deploy	Docker, Kubernetes, Ansible
Operate	AWS, Azure, GCP
Monitor	Prometheus, Grafana, ELK Stack

- DevOps = Collaboration + Automation + Continuous Delivery.
- Different from Agile because it includes Operations in the process.
- Benefits: Speed, quality, teamwork.
- Lifecycle: Plan → Code → Build → Test → Release → Deploy → Operate → Monitor.
- Principles: CAMS.
- Toolchain = tools for each stage of the lifecycle.

2.1 Linux Basics: Commands, File System, Permissions

Why Linux?

- Most servers & cloud instances use Linux.
- Powerful command-line control.
- Open-source and free.

Basic Commands

Command	Purpose	Example
pwd	Show current directory	pwd
ls	List files	ls -l (long format)
cd	Change directory	cd /home/user
mkdir	Create folder	mkdir devops
rm	Delete files	rm file.txt
cp	Copy files	cp a.txt b.txt
mv	Move/rename	mv file.txt docs/
cat	View file contents	cat notes.txt
nano / vi	Edit files	nano config.txt
history	See previous commands	history

Linux File System

- **/ (root)** – Top of the file system
- **/home** – User files

- **/etc** – Config files
- **/var** – Logs, temporary data
- **/bin** – Basic commands
- **/usr** – Installed programs

Analogy: Think of Linux like a city:

- / = city gate
- /home = houses
- /etc = government offices (config)
- /var = post office (logs)

Permissions

- **Three Types**: Read (r), Write (w), Execute (x)
- **Three Groups**: Owner, Group, Others
- **View permissions**:
- ls -l
  Output example:

  -rw-r--r-- 1 user user 1234 Aug 14 file.txt
- **Change permissions**:
- chmod 755 script.sh

Quick Tip:
chmod 755 → Owner: all rights, Others: read & execute only.

2.2 SSH & Remote Access

Why? – DevOps engineers often manage remote servers.

- **Connect to a remote server**:
- ssh username@server_ip
- **Copy files to/from server**:
- scp local.txt user@server:/home/user/
- **Generate SSH key for passwordless login**:
- ssh-keygen
- ssh-copy-id user@server

2.3 Process Management

- **List running processes**:
- ps aux
- top
- htop
- **Kill a process**:
- kill <PID>

- **Find a process**:
- pgrep nginx

Example: Restarting a crashed web server:

```
sudo systemctl restart nginx
```

2.4 Networking Basics in Linux

Command	Purpose
ifconfig / ip addr	Show network interfaces
ping google.com	Test connectivity
netstat -tuln	Show open ports
curl	Test APIs / download data
wget	Download files
nslookup	Check DNS

2.5 Shell Scripting for Automation

Why? – In DevOps, you automate repetitive tasks.

Example: Backup script

```
#!/bin/bash
DATE=$(date +%F)
tar -czf backup_$DATE.tar.gz /home/user/data
echo "Backup completed on $DATE"
```

- Save as backup.sh
- Make executable:
- chmod +x backup.sh
- Run:
- ./backup.sh

Shell scripting use cases in DevOps:

- Automating deployments
- Cleaning logs
- Monitoring system health
- Scheduling tasks (cron jobs)

End of Chapter Summary:

- Learn basic Linux commands & file structure.
- Understand permissions (chmod, chown).
- Use SSH for remote server access.
- Manage processes & check system resources.

- Know basic networking commands.
- Write simple shell scripts for automation.

3.1 What is Version Control?

Version control is a system that **tracks changes** to your code over time, so you can:

- Revert to a previous version if something breaks
- Work with others without overwriting each other's changes
- Keep a history of every modification

Real-life analogy:
Think of it like **Google Docs for code** — you can see changes, restore older versions, and work with friends without chaos.

3.2 Git Basics

Git is the most popular version control system today.

Installation

- **Linux:**
- sudo apt install git
- **Windows/Mac:** Download from git-scm.com

Basic Git Commands

Command	Purpose	Example
git init	Start a new repository	git init
git clone <url>	Copy an existing repo	git clone https://github.com/user/repo.git
git status	Check changes	git status
git add <file>	Stage changes	git add index.html
git commit -m "msg"	Save changes	git commit -m "Added home page"
git push	Upload to remote repo	git push origin main
git pull	Get latest changes	git pull origin main
git merge <branch>	Merge branch into current	git merge dev

3.3 Branching & Merging Strategies

- **Branch:** A separate line of development.
- **Merge:** Combine changes from one branch into another.

Popular Strategies:

1. **Feature Branching** – Create a new branch for every feature:
2. git checkout -b feature-login
3. **Git Flow** – Structured workflow:
    - main → production
    - develop → staging
    - feature/*, release/*, hotfix/* for specific purposes

3.4 Git Workflows

1. **Centralized Workflow** – Everyone works on a single branch (not recommended for big teams).
2. **Feature Branch Workflow** – Each feature gets its own branch → merge when done.
3. **Git Flow Workflow** – For big projects with release cycles.
4. **Forking Workflow** – Used in open-source projects.

3.5 Resolving Merge Conflicts

Why conflicts happen:
Two people edit the same part of a file and Git doesn't know which change to keep.

Steps to fix:

1. Open the file with conflict → Look for:
2. <<<<<<< HEAD
3. Your changes
4. =======
5. Their changes
6. >>>>>>> branch-name
7. Keep the correct code, remove the markers.
8. Add and commit:
9. git add file.txt
10. git commit -m "Resolved conflict"

3.6 GitHub / GitLab / Bitbucket Usage

- **GitHub** → Most popular, used in open source
- **GitLab** → Built-in CI/CD
- **Bitbucket** → Integrates well with Jira

Common tasks:

- Creating repositories online
- Forking projects
- Pull Requests (PRs) / Merge Requests (MRs)
- Code reviews

3.7 Git Hooks & Automation

Git Hooks: Scripts that run automatically when certain Git events happen.

Example: Pre-commit hook to check code formatting:

```
#!/bin/bash
echo "Checking code formatting..."
npm run lint
```

Save it in `.git/hooks/pre-commit` and make it executable:

```
chmod +x .git/hooks/pre-commit
```

Use cases:

- Auto-format code before commit
- Prevent commits with sensitive data
- Run tests before pushing code

- Git is a tool for tracking code changes.
- Learn init → add → commit → push → pull → merge.
- Use branching strategies to keep work organized.
- Merge conflicts are normal — fix them by editing files and committing.
- GitHub/GitLab/Bitbucket are remote platforms for collaboration.
- Git Hooks can automate checks and tasks.

4.1 Why Build Tools Are Needed

When developers write code, it's often split into multiple files, libraries, and dependencies. Before that code can run in production, it usually needs to be:

- **Compiled** (if it's in languages like Java, C++, etc.)
- **Packaged** (into a format that can be deployed, e.g., .jar, .war, .zip)
- **Linked** with required libraries
- **Tested** automatically
- **Deployed** to a server

Real-life analogy:
Think of it like baking a cake:

- Ingredients = source code + dependencies
- Recipe = build script
- Oven = build tool
  Without a build tool, you'd have to manually mix, measure, and bake every time — which is slow and error-prone.

Benefits of build tools:
Automate repetitive tasks
Ensure consistency
Save time during CI/CD
Reduce human errors

4.2 Maven Basics

Type: Java build tool
Main file: pom.xml (Project Object Model)
Common uses:

- Compile Java code
- Download dependencies
- Run tests
- Package into .jar or .war

Basic Maven Commands:

```
mvn clean      # Remove old build files
mvn compile    # Compile the code
mvn test       # Run tests
mvn package    # Create a package
mvn install    # Install package into local repository
```

4.3 Gradle Basics

Type: Flexible build tool for Java, Kotlin, Groovy, and more
Main file: build.gradle (or build.gradle.kts for Kotlin DSL)
Advantages over Maven: Faster builds, less XML, more scripting flexibility

Basic Gradle Commands:

```
gradle build      # Compile and package
gradle clean      # Clean old builds
gradle test       # Run tests
gradle run        # Run the application
```

4.4 npm & Yarn Basics

Type: Build and package managers for JavaScript/Node.js projects

npm Commands:

```
npm init       # Create package.json
npm install    # Install dependencies
npm run build  # Run build script
npm test       # Run tests
```

Yarn Commands:

```
yarn init      # Create package.json
yarn install   # Install dependencies
yarn build     # Build the project
yarn test      # Run tests
```

Note: Yarn is often faster and more secure, but npm is more widely used.

4.5 Automated Builds in CI/CD

In **CI/CD pipelines**, build tools are integrated to:

1. **Automatically compile** code whenever a developer pushes changes
2. **Run automated tests** after the build
3. **Generate artifacts** (packages, containers) ready for deployment

Example with Jenkins:

```
pipeline {
    stages {
        stage('Build') {
            steps {
                sh 'mvn clean package'
            }
        }
        stage('Test') {
            steps {
                sh 'mvn test'
            }
        }
    }
}
```

Outcome:
If the build or tests fail → CI pipeline stops → Developers fix before deployment.

- Build tools automate compiling, packaging, and testing.
- **Maven & Gradle** → Java ecosystem.
- **npm & Yarn** → JavaScript/Node.js.

- In CI/CD, build tools are essential for automation and quality assurance.

Feature / Tool	Maven	Gradle	npm / yarn
Primary Use	Java-based project build & dependency management	Flexible build automation for many languages (Java, Kotlin, etc.)	JavaScript/Node.js dependency & build management
Language	XML (pom.xml)	Groovy or Kotlin DSL	JSON (package.json)
Performance	Slower (XML parsing, no incremental builds)	Faster with incremental builds & caching	Fast (yarn is generally faster than npm)
Flexibility	Convention over configuration, limited flexibility	Highly flexible, supports complex builds	Mostly for JS projects, limited to Node ecosystem
Dependency Management	Central Maven Repository	Maven & Ivy repositories	npm registry
Learning Curve	Easy to learn (structured XML)	Steeper (Groovy/Kotlin scripting)	Easy for JS developers
Community Support	Large, mature	Large, growing	Very large (JS ecosystem)
Use in CI/CD	Widely supported in Jenkins, GitHub Actions, etc.	Widely supported, better performance	Widely used in JS CI/CD pipelines
Best For	Enterprise Java projects	Complex, multi-language builds	JavaScript/Node.js projects

Concept of CI

- **Definition:** Continuous Integration is a development practice where developers frequently merge their code changes into a shared repository, usually multiple times a day.
- **Purpose:**
  - Detect integration issues early
  - Automate build and testing processes
  - Improve code quality and team productivity
- **Key Steps:**
  1. Developer pushes code to version control (GitHub/GitLab).
  2. CI server automatically builds and tests the code.
  3. Feedback is sent to developers instantly.

Jenkins Fundamentals

- **Jenkins:** An open-source automation server used for building, testing, and deploying applications.
- **Features:**
  - Supports 1000+ plugins (Git, Docker, Maven, etc.)
  - Works on pipelines and freestyle jobs
  - Can be integrated with almost any technology stack
- **Basic Workflow:**
  1. Configure a **Job** (freestyle or pipeline)
  2. Link it with source code repository
  3. Set triggers (manual, scheduled, or webhook)
  4. Build, test, and deploy

Jenkins Pipelines

- **Scripted Pipeline:**
  - Written in Groovy
  - More flexible but complex
  - Example:

```
node {
    stage('Build') { echo 'Building...' }
    stage('Test') { echo 'Testing...' }
    stage('Deploy') { echo 'Deploying...' }
}
```

- **Declarative Pipeline:**
  - More structured and beginner-friendly
  - Example:

```
pipeline {
    agent any
    stages {
        stage('Build') { steps { echo 'Building...' } }
        stage('Test') { steps { echo 'Testing...' } }
        stage('Deploy') { steps { echo 'Deploying...' } }
    }
}
```

GitHub Actions / GitLab CI

- **GitHub Actions:**
  - Built-in CI/CD for GitHub repositories
  - Uses YAML configuration
  - Example:

```
name: CI
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - run: echo "Building project..."
```

- **GitLab CI:**
  - Integrated CI/CD with GitLab repositories
  - Uses .gitlab-ci.yml file for configuration
  - Example:

```
stages:
  - build
build-job:
  stage: build
  script:
    - echo "Building project..."
```

CI Best Practices

1. **Commit frequently** to detect issues early.
2. **Automate builds** to reduce manual errors.
3. **Run tests on every build** to ensure quality.
4. **Keep pipelines fast** for quicker feedback.
5. **Use separate environments** for build, test, and production.
6. **Fail early and visibly** so the team can respond quickly.

6.1 Concept

Continuous Testing is the practice of running automated tests **throughout the software delivery pipeline** to get instant feedback on code quality, functionality, and performance. It ensures that **every code change** is tested before moving to the next stage.

6.2 Automated Testing in DevOps

- **Goal**: Detect defects as early as possible.
- **Key idea**: Integrate testing with CI/CD so tests run automatically after code changes.
- **Benefits**:
  - Faster feedback loop.
  - Improved software quality.
  - Reduced manual effort.

6.3 Testing Types

Type	Purpose	When Performed	Tools
Unit Testing	Test individual code components (functions, methods).	Early in development (developer stage).	JUnit, PyTest, NUnit
Integration Testing	Test how different modules work together.	After unit tests, before full app test.	TestNG, Postman, PyTest
End-to-End (E2E) Testing	Test the complete workflow from start to finish.	After integration tests, before release.	Selenium, Cypress, Playwright

6.4 Popular Tools

Tool	Language/Use	Best For
Selenium	Multi-language, browser automation	E2E UI testing
JUnit	Java	Unit testing
PyTest	Python	Unit + integration testing
TestNG	Java	Integration + functional tests
Cypress	JavaScript	E2E + frontend testing

6.5 Testing in CI/CD Pipelines

- **Where it fits**:
  - **After build** → Run unit & integration tests.
  - **Before deployment** → Run E2E tests.
- **Implementation steps**:
  1. Add test scripts in repository.
  2. Configure pipeline to run tests automatically.
  3. Generate and store test reports.
  4. Fail the pipeline if tests fail.

Example **Jenkins pipeline snippet** for testing:

```
stage('Test') {
   steps {
      sh 'pytest --junitxml=reports/results.xml'
   }
}
```

7.1 Continuous Delivery vs Continuous Deployment

Feature	Continuous Delivery	Continuous Deployment
Definition	Code is always in a deployable state, but deployment to production is **manual**	Every code change that passes tests is **automatically deployed** to production
Automation Level	High automation in build/test, manual approval for release	Fully automated, no manual intervention
Risk	Lower risk due to manual approval	Higher risk if automated tests are insufficient
Example	Code is merged, QA approves, then deployed	Code is merged and instantly deployed

7.2 Deployment Strategies

1. **Blue-Green Deployment** – Maintain two environments (**Blue** - live, **Green** - idle). Deploy to Green, test, then switch traffic.
2. **Rolling Updates** – Gradually replace old versions with new ones without downtime.
3. **Canary Releases** – Release to a small subset of users first, then expand if no issues are found.

7.3 CI/CD Tools for Deployment

- **Jenkins** – Highly customizable automation server.
- **GitHub Actions** – Integrated with GitHub repositories, YAML-based workflows.
- **GitLab CI** – Built-in CI/CD with GitLab projects.

7.4 Artifact Repositories

- **Nexus Repository** – Stores build artifacts like JAR, WAR, Docker images.
- **JFrog Artifactory** – Universal artifact management supporting multiple formats (Docker, Maven, npm)

8.1 Introduction to Containers & Virtualization

- **Virtualization**: Creates virtual machines (VMs) with their own OS running on a hypervisor (e.g., VMware, VirtualBox).
- **Containerization**: Packages applications and dependencies together, sharing the host OS kernel.
- **Key Difference**:
    - VMs → Heavy, slower startup, full OS.
    - Containers → Lightweight, faster, portable.

- **Benefits of Containers**:
  - Consistent environment across development, testing, and production.
  - Faster deployments.
  - Better resource utilization.

8.2 Docker Basics: Images, Containers, Volumes, Networks

- **Docker Image**: Read-only template with application code + dependencies.
- **Docker Container**: Running instance of a Docker image.
- **Volumes**: Persistent storage mechanism for containers.
- **Networks**: Enable communication between containers and the outside world.
- **Basic Commands**:
  - `docker pull nginx  # Download image`
  - `docker run -d -p 8080:80 nginx # Run container`
  - `docker ps  # List running containers`
  - `docker stop <container_id>  # Stop container`

8.3 Dockerfile Creation & Optimization

- **Dockerfile**: Script with instructions to build an image.
- **Example**:

```
FROM python:3.10-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "app.py"]
```

- **Optimization Tips**:
  - Use smaller base images (e.g., `alpine`).
  - Combine commands to reduce image layers.
  - Use `.dockerignore` to skip unnecessary files.

8.4 Docker Compose

- Tool to define and manage multi-container applications.
- **Example** (`docker-compose.yml`):

```
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
  db:
    image: postgres:14
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: pass
```

- **Command**:
    - docker-compose up -d

8.5 Docker Registry (Docker Hub, Private Registry)

- **Docker Hub**: Public registry for sharing images.
- **Private Registry**: Host your own registry for internal use.
- **Usage**:
    - docker tag myapp:latest myusername/myapp:v1
    - docker push myusername/myapp:v1
    - docker pull myusername/myapp:v1

9.1. Introduction to Kubernetes

- **Kubernetes (K8s)** is an open-source platform for automating the deployment, scaling, and management of containerized applications.
- It groups containers into **Pods** and manages them across a cluster of machines.
- Key benefits: automation, scalability, self-healing, load balancing.

9.2. Kubernetes Architecture

Kubernetes has **two main components**:

1. **Control Plane** – Manages the cluster.
    - **API Server** – Entry point for all administrative tasks.
    - **Scheduler** – Decides where pods should run.
    - **Controller Manager** – Ensures desired state matches actual state.
    - **etcd** – Stores cluster configuration data.
2. **Worker Nodes** – Run the applications.
    - **Kubelet** – Ensures pods are running as expected.
    - **Kube-proxy** – Manages networking between pods and services.
    - **Container Runtime** – Runs the containers (e.g., Docker, containerd).

9.3. Pods, Services, Deployments, ReplicaSets

- **Pod** – Smallest deployable unit in Kubernetes (contains one or more containers).
- **Service** – Provides network access to a set of pods.
- **Deployment** – Manages multiple replicas of pods, supports rolling updates.
- **ReplicaSet** – Ensures the desired number of pod replicas are running.

9.4. ConfigMaps & Secrets

- **ConfigMap** – Stores configuration data in key-value pairs for pods.
- **Secret** – Stores sensitive information like passwords, API keys (base64 encoded).

9.5. Scaling & Auto-healing

- **Scaling** – Add or remove pod replicas based on demand (Horizontal Pod Autoscaler).
- **Auto-healing** – Kubernetes automatically restarts failed pods or reschedules them if a node fails.

9.6. Helm Charts for App Deployment

- **Helm** – A package manager for Kubernetes.
- **Helm Charts** – Pre-configured Kubernetes resources packaged together, making deployment easier and reusable.

9.7. OpenShift Basics

- **OpenShift** – A Kubernetes-based container platform by Red Hat.
- Adds features like:
    - Developer-friendly web console
    - Built-in CI/CD pipeline
    - Enhanced security policies

10.1. IaC Concepts & Benefits

- **Definition**: IaC is the practice of managing and provisioning infrastructure through machine-readable configuration files rather than manual processes.
- **Benefits**:
    - **Consistency** – No "it works on my machine" issues.
    - **Version Control** – Infra changes are tracked like code.
    - **Automation** – Faster deployments, fewer manual errors.
    - **Scalability** – Easy to replicate environments.
    - **Rollback** – Revert to previous configurations quickly.

10.2. Terraform Basics

Terraform is a **declarative IaC tool** from HashiCorp.

- **Providers**: Plugins that allow Terraform to interact with various platforms (AWS, Azure, GCP, Kubernetes).
- **Resources**: Define components like VMs, networks, databases.
- **Variables**: Store configurable values for reusability.
- **Outputs**: Share values after deployment (e.g., public IP).
- **Workflow**:
    1. `terraform init` – Initialize working directory.
    2. `terraform plan` – Preview changes.
    3. `terraform apply` – Create/update resources.

4. terraform destroy – Remove resources.

10.3. Ansible Basics

Ansible is a **configuration management tool** (procedural style).

- **Playbooks**: YAML files describing automation steps.
- **Roles**: Structured way to organize playbooks.
- **Modules**: Pre-built commands to manage resources.
- **Advantages**:
  - Agentless (uses SSH).
  - Easy syntax.
  - Works for both cloud & on-prem systems.

10.4. AWS CloudFormation Basics

- AWS-native IaC service.
- Templates written in **YAML or JSON**.
- Supports **Stacks** (a group of resources managed together).
- Integration with **AWS Services** like EC2, S3, RDS.

10.5. CI/CD Integration with IaC

- **Terraform in CI/CD**: Validate and apply infra changes automatically through pipelines (Jenkins, GitHub Actions, GitLab CI).
- **Ansible in CI/CD**: Automate server configuration post-deployment.
- **Benefits**:
  - Infrastructure is tested like application code.
  - Faster and more reliable provisioning.
  - Eliminates manual intervention in production releases.

11. Cloud Computing for DevOps

Cloud computing plays a crucial role in DevOps by providing on-demand infrastructure, scalability, and flexibility to deploy and manage applications faster. It eliminates the need for heavy upfront investment in hardware and allows teams to focus on automation, continuous delivery, and scalability.

Overview of AWS, Azure, and GCP

- **Amazon Web Services (AWS)** – The most widely used cloud platform with a rich ecosystem for compute (EC2, ECS, Lambda), storage (S3, EBS), databases (RDS, DynamoDB), and DevOps tools (CodePipeline, CodeBuild).

- **Microsoft Azure** – Strong integration with Microsoft technologies, Azure DevOps, Azure Kubernetes Service (AKS), and Azure Functions for serverless computing.
- **Google Cloud Platform (GCP)** – Known for data analytics, AI/ML services, Kubernetes Engine (GKE), and cost-effective infrastructure solutions.

Cloud Services for DevOps

- **Compute** – Virtual machines (EC2, Azure VMs, GCE), containers (ECS, AKS, GKE), and serverless compute (AWS Lambda, Azure Functions, Google Cloud Functions).
- **Storage** – Object storage (S3, Blob Storage, Cloud Storage), block storage, and archival storage for backups.
- **Networking** – Virtual networks, load balancers, VPNs, and Content Delivery Networks (CDNs) for global distribution.

Deploying Applications to the Cloud

- Use **Infrastructure as Code (IaC)** tools like Terraform, AWS CloudFormation, or Azure Resource Manager templates for consistent provisioning.
- Automate deployments using CI/CD pipelines integrated with cloud platforms.
- Implement rolling updates, blue-green deployments, or canary releases to minimize downtime.

Serverless Basics

- **AWS Lambda** – Event-driven compute service that runs code without managing servers, ideal for microservices, automation scripts, and APIs.
- **Azure Functions** – Similar to Lambda but tightly integrated into Azure services.
- **Advantages** – Cost efficiency (pay only for execution time), scalability, and reduced infrastructure management.
- **Use Cases** – Event processing, scheduled tasks, lightweight APIs, and real-time data processing.

12. Monitoring & Logging

Monitoring and logging are crucial for ensuring that applications and infrastructure are healthy, performant, and reliable. They help DevOps teams detect issues early, understand system behavior, and maintain service-level objectives (SLOs).

Importance of Monitoring in DevOps

- **Early Detection of Issues** – Identify anomalies before they impact end users.
- **Performance Tracking** – Measure CPU, memory, network, and application metrics.
- **Capacity Planning** – Use historical data to predict and allocate resources.
- **Compliance & Audit** – Maintain logs for security audits and compliance.
- **Feedback Loop** – Provide insights for continuous improvement.

Prometheus & Grafana Basics

- **Prometheus**: An open-source monitoring system that scrapes and stores time-series metrics.
    - Features: Pull-based metrics collection, flexible queries, alerting.
    - Example: Monitoring CPU usage of Kubernetes pods.
- **Grafana**: A visualization and dashboard tool that integrates with Prometheus (and other data sources) to display metrics in real-time.
    - Features: Custom dashboards, alert rules, multiple data sources.

ELK Stack (Elasticsearch, Logstash, Kibana)

- **Elasticsearch** – Stores and indexes log data for fast searching.
- **Logstash** – Collects, parses, and processes logs from various sources.
- **Kibana** – Visualizes log data from Elasticsearch through dashboards.
- **Use Case** – Centralized logging for microservices in a distributed environment.

Alerting & Incident Management

- **Alerting Tools**: Prometheus Alertmanager, PagerDuty, Opsgenie.
- **Best Practices**:
    - Define thresholds for key metrics.
    - Avoid alert fatigue with actionable alerts only.
    - Integrate alerts into incident response workflows.

Application Performance Monitoring (APM) Tools

- **Examples**: New Relic, Datadog, AppDynamics.
- **Capabilities**:
    - Trace requests through distributed systems.
    - Detect bottlenecks and slow database queries.
    - Measure transaction response times.

Tip: In modern DevOps, combine **real-time monitoring** (Prometheus + Grafana) with **centralized logging** (ELK Stack) for complete observability.

13. Security in DevOps (DevSecOps)

Importance of DevSecOps

- **Definition**: DevSecOps integrates security practices into the DevOps process from the start rather than as an afterthought.
- **Goal**: Shift security "left" in the SDLC to catch and fix vulnerabilities early.

- **Benefits**:
    - Reduced risk of breaches.
    - Faster remediation of vulnerabilities.
    - Improved compliance.

13.1 DevSecOps Concepts

- **Shift-Left Security** – Involve security teams during development, not after deployment.
- **Security Automation** – Use automated tools to scan code, dependencies, and containers.
- **Continuous Security Testing** – Integrate security checks in every CI/CD stage.
- **Threat Modeling** – Identify potential security risks before coding begins.
- **Security as Code** – Define and manage security policies through code for reproducibility.

13.2 Security Scanning Tools

- **SonarQube** – Detects code quality issues and security vulnerabilities in multiple languages.
- **Snyk** – Scans for vulnerabilities in open-source dependencies and container images.
- **Trivy** – Lightweight vulnerability scanner for container images, filesystems, and Git repositories.

13.3 Secrets Management

- **Why?** – Storing secrets (API keys, passwords) in code is risky; they should be encrypted and managed securely.
- **Tools**:
    - **HashiCorp Vault** – Secure storage and access to secrets, encryption keys, and credentials.
    - **AWS Secrets Manager** – Managed service to store and rotate secrets automatically.
- **Best Practices**:
    - Avoid hardcoding secrets in code.
    - Implement role-based access control.
    - Rotate keys regularly.

13.4 Secure CI/CD Pipelines

- Integrate **static application security testing (SAST)** early in the build stage.
- Perform **dynamic application security testing (DAST)** in staging environments.
- Scan container images before pushing them to registries.
- Use **signed artifacts** to prevent tampering.
- Implement **least privilege access** for build agents and deployment systems.

13.5 Compliance & Governance

- **Compliance Standards**:
  - GDPR, HIPAA, ISO 27001, PCI-DSS depending on the industry.
- **Governance**:
  - Centralized policy management.
  - Regular audits and penetration testing.
  - Maintain audit logs for all deployments and changes.

Quick Tip: DevSecOps isn't just about tools — it's about culture. Every developer, tester, and ops engineer should own security responsibilities.

14. Networking & System Design for DevOps

- **Basics of Networking** – Understanding TCP/IP layers, DNS resolution process, HTTP/HTTPS protocols, IP addressing, subnetting, NAT, firewalls.
- **Load Balancing** – Concepts of round robin, least connections, IP hash; hardware vs. software load balancers; session persistence.
- **API Gateways** – Role in routing, authentication, throttling, caching, monitoring, and transformation of requests (e.g., Kong, AWS API Gateway).
- **Reverse Proxy (NGINX, HAProxy)** – How reverse proxies improve security, scalability, and performance; SSL termination; caching strategies.
- **CDN Basics** – Content distribution, edge caching, reducing latency, popular CDNs (Cloudflare, Akamai, AWS CloudFront).
- **Microservices Architecture** – Principles of microservices, service discovery, inter-service communication (REST, gRPC, message queues), handling failures, scaling services independently.
- **High Availability & Fault Tolerance** – Active-active vs. active-passive setups, redundancy, failover mechanisms.
- **Monitoring & Logging in Distributed Systems** – Centralized logging, tracing (Jaeger, Zipkin), service meshes (Istio, Linkerd)

15. Advanced DevOps Topics

- **GitOps (ArgoCD, FluxCD)** – Managing infrastructure and applications using Git as the single source of truth, enabling automated CI/CD pipelines.
- **Chaos Engineering (Gremlin, Litmus)** – Intentionally introducing failures to test system resilience and recovery strategies.
- **Site Reliability Engineering (SRE) Basics** – Applying software engineering principles to infrastructure and operations to improve reliability and uptime.
- **Service Mesh (Istio, Linkerd)** – Abstracting service-to-service communication for observability, traffic control, and security in microservices.
- **Kubernetes Operators** – Automating the management of complex Kubernetes applications using custom controllers.

- **Hybrid & Multi-cloud DevOps** – Designing and managing deployments across multiple cloud providers or mixing on-premise and cloud infrastructure.
- **Cost Optimization in Cloud** – Strategies and tools to reduce cloud expenditure without compromising performance.

16. Real-World DevOps Projects

- End-to-End CI/CD for a Web App using Jenkins + Docker + Kubernetes + AWS
- Deploying Microservices on Kubernetes with Helm & Istio
- Automated Infrastructure Provisioning with Terraform & Ansible
- Monitoring a Production App with Prometheus + Grafana
- Securing a Pipeline with DevSecOps tools

17. Soft Skills & Best Practices

- Agile/Scrum in DevOps
- Collaboration Tools (Slack, MS Teams)
- Incident & Problem Management
- Documentation in DevOps
- Continuous Learning Culture