Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

AWS NETWORKING

AWS VPC (VIRTUAL PRIVATE CLOUD) BASICS – STUDY MATERIAL

INTRODUCTION TO AWS VPC

What is AWS VPC?

AWS **Virtual Private Cloud (VPC)** is a **logically isolated** network in AWS that allows users to launch and manage AWS resources securely. It functions like a **private data center in the cloud,** providing control over networking, security, and internet access.

Key Features of AWS VPC

✓ **Logical Network Isolation** – Users can define their own IP address ranges.
✓ **Subnetting** – Allows creation of **public and private subnets** for different workloads.
✓ **Internet Gateway (IGW)** – Enables internet access for resources inside VPC.
✓ **Security Groups & Network ACLs** – Control inbound and outbound traffic.
✓ **Route Tables** – Define how network traffic is directed.

✓ **VPN & Direct Connect** – Securely connect on-premises networks to AWS.

CHAPTER 1: AWS VPC ARCHITECTURE & COMPONENTS

1. Key Components of AWS VPC

Component	Description
VPC (Virtual Private Cloud)	A logically isolated network in AWS.
Subnets	Smaller network divisions within a VPC (Public & Private Subnets).
Internet Gateway (IGW)	Allows internet access for public subnets.
NAT Gateway	Enables private subnet instances to access the internet.
Route Tables	Define how traffic flows between subnets and the internet.
Security Groups	Controls inbound/outbound traffic for EC2 instances.
Network ACLs (NACLs)	Stateless firewall controlling subnet-level traffic.
VPC Peering	Connects two VPCs for private communication.
VPN & Direct Connect	Securely connects AWS VPC with on-premises networks.

📌 **Example:**
A **web application** may use a **public subnet for a web server** and a **private subnet for a database** to ensure security.

Chapter 2: Creating an AWS VPC Step-by-Step

Step 1: Create a VPC

1. Open **AWS Console** → Navigate to **VPC Dashboard**.
2. Click **"Create VPC"**.
3. **VPC Name:** MyAppVPC.
4. **IPv4 CIDR Block:** 10.0.0.0/16 (Allows 65,536 IPs).
5. Click **"Create VPC"**.

📌 **Expected Outcome:**
A **new VPC** is created with a **private IP range** (10.0.0.0/16).

Step 2: Create Subnets

1. Open **VPC Console** → Click **"Subnets"** → **Create Subnet**.
2. **Subnet 1 (Public Subnet):**
   - Name: Public-Subnet.
   - CIDR Block: 10.0.1.0/24.
   - Enable **Auto-Assign Public IPv4**: **Yes**.
3. **Subnet 2 (Private Subnet):**
   - Name: Private-Subnet.
   - CIDR Block: 10.0.2.0/24.

- Enable Auto-Assign Public IPv4: **No**.

📌 **Expected Outcome:**

Two subnets are created:

✓ **Public Subnet (10.0.1.0/24)** – Internet access enabled.

✓ **Private Subnet (10.0.2.0/24)** – No direct internet access.

Step 3: Attach an Internet Gateway (IGW)

1. Navigate to **VPC Console** → **Internet Gateways**.
2. Click **"Create Internet Gateway"** → Name it MyIGW.
3. Click **"Attach to VPC"** → Select MyAppVPC.

📌 **Expected Outcome:**

The **Internet Gateway is attached,** allowing public subnet resources to access the internet.

Step 4: Configure Route Tables

1. Navigate to **VPC Console** → **Route Tables**.
2. **Public Route Table**:
   - Select Public Route Table → Click **"Edit Routes"**.
   - Add **0.0.0.0/0** → Target: **Internet Gateway (MyIGW)**.
3. **Private Route Table**:
   - Used for internal traffic (no internet access).

📌 **Expected Outcome:**

✓ Public Subnet → **Can access the internet** via **Internet Gateway**.

✓ Private Subnet → **No direct internet access** (for database security).

Step 5: Add a NAT Gateway (For Private Subnet Internet Access)

1. Navigate to **VPC Console** → **NAT Gateways** → Click **"Create NAT Gateway"**.
2. **Select Public Subnet** (10.0.1.0/24).
3. **Elastic IP:** Allocate a new **Elastic IP (EIP)**.
4. Click **Create NAT Gateway**.
5. **Update Private Route Table**:
    - Go to **Route Tables** → Select **Private Route Table**.
    - Add **0.0.0.0/0** → Target: **NAT Gateway**.

📌 **Expected Outcome:**
✓ **Private Subnet instances can access the internet** but **remain hidden** from incoming connections.

CHAPTER 3: CONFIGURING SECURITY GROUPS & NETWORK ACLS

1. Creating a Security Group for Web Server

1. Navigate to **EC2 Console** → **Security Groups**.
2. Click **"Create Security Group"**.
3. **Security Group Name:** WebServerSG.
4. **Inbound Rules:**
    - Allow HTTP (80) and HTTPS (443) from **0.0.0.0/0**.
    - Allow SSH (22) only from **your IP**.

📌 **Expected Outcome:**
The **Web Server can receive HTTP traffic but restricts SSH to specific IPs**.

---

2. Creating a Security Group for Database

1. **Security Group Name:** DatabaseSG.
2. **Inbound Rules:**
    - Allow MySQL (3306) traffic **only from WebServerSG**.

📌 **Expected Outcome:**
The **Database is secured** and **accessible only from the Web Server**.

---

3. Configuring Network ACLs (Optional Extra Security)

1. Navigate to **VPC Console → Network ACLs**.
2. Click **Create Network ACL** → Associate it with **MyAppVPC**.
3. Configure **Inbound Rules**:
    - Allow HTTP (80) and HTTPS (443).
    - Allow SSH (22) from your IP.
4. Configure **Outbound Rules**:
    - Allow all traffic (0.0.0.0/0).

📌 **Expected Outcome:**
A **stateless firewall** adds **extra security** to subnets.

CHAPTER 4: CONNECTING TO EC2 INSTANCE INSIDE THE VPC

1. **Launch an EC2 instance** in the **Public Subnet**.

2. **Assign WebServerSG security group** to allow web traffic.
3. **Connect to EC2 via SSH**:
4. ssh -i my-key.pem ec2-user@your-public-ip
5. **Check Internet Connectivity**:
6. curl https://www.google.com

📌 **Expected Outcome:** The **public EC2 instance can access the internet**.

Chapter 5: AWS VPC Best Practices

✓ **Use Public Subnets for Web Servers and Private Subnets for Databases**.
✓ **Restrict SSH Access to Your IP** using Security Groups.
✓ **Use NAT Gateway for Private Subnet Internet Access**.
✓ **Enable VPC Flow Logs for Network Monitoring**.
✓ **Use AWS Direct Connect for Secure On-Premise Connectivity**.

📌 **Example:**
A **finance company** enforces **strict security policies** using **private subnets for databases and public subnets for web applications**.

Conclusion: Mastering AWS VPC

By following this guide, you have:
✅ **Created a VPC and Subnets**.
✅ **Configured Internet and NAT Gateways**.
✅ **Implemented Security Groups and Route Tables**.
✅ **Launched an EC2 instance inside the VPC**.

FINAL EXERCISE:

1. **Create a second VPC and set up VPC Peering between two VPCs**.
2. **Enable AWS VPC Flow Logs** to monitor network traffic.
3. **Set up a Site-to-Site VPN to connect an on-premises network to AWS**.

AWS Route 53 (Domain Name System) – Study Material

Introduction to AWS Route 53

What is AWS Route 53?

AWS **Route 53** is a highly scalable and reliable **Domain Name System (DNS) web service** that helps manage domain registration, DNS routing, and health checking of AWS-hosted applications. It translates **human-readable domain names** (e.g., www.mywebsite.com) into **IP addresses** (e.g., 192.168.1.1) that computers use to communicate.

Key Features of AWS Route 53

✓ **Domain Registration** – Buy and manage domain names directly from AWS.
✓ **DNS Routing** – Directs internet traffic to AWS resources or external servers.
✓ **Traffic Routing Policies** – Supports **latency-based, geolocation, failover, and weighted routing**.
✓ **Health Checks & Failover** – Monitors application health and redirects traffic when failures occur.
✓ **Highly Available & Scalable** – AWS Route 53 is a **global DNS service** that provides **99.99% uptime**.
✓ **Secure & Compliant** – Integrated with **AWS IAM, DNSSEC, and AWS Shield** for security.

Chapter 1: AWS Route 53 Architecture & Key Concepts

1. AWS Route 53 Components

Component	Description
Hosted Zones	A collection of DNS records for a domain.
DNS Records	Defines how traffic is routed (A, CNAME, MX, TXT, etc.).
Traffic Policies	Controls how DNS queries are routed.
Health Checks	Monitors application availability & failover.
Domain Registration	Allows users to register new domains within AWS.

📌 **Example:**

A **website (www.myapp.com) hosted on EC2** uses Route 53 to map its domain name to an **Elastic Load Balancer (ALB) IP address**.

CHAPTER 2: REGISTERING A DOMAIN WITH AWS ROUTE 53

Step 1: Purchase a Domain Name

1. Open **AWS Console** → Navigate to **Route 53**.
2. Click **"Registered Domains"** → Click **"Register Domain"**.
3. Enter the **desired domain name** (e.g., myapp.com).
4. Select **domain extension** (e.g., .com, .org, .net).
5. Provide **contact details** (email, phone).
6. Click **"Purchase"** and wait for domain registration confirmation.

📌 **Expected Outcome:**

Your domain is **registered and ready to use** in Route 53.

Step 2: Create a Hosted Zone

1. Open **Route 53 Console** → Click **"Hosted Zones"**.
2. Click **"Create Hosted Zone"**.
3. Enter the **Domain Name** (myapp.com).
4. Choose **Public Hosted Zone** (for internet-facing websites).
5. Click **"Create"**.

📌 **Expected Outcome:**
A **new Hosted Zone** is created with **default NS (Name Server) and SOA (Start of Authority) records**.

Chapter 3: Configuring DNS Records in Route 53

1. Common DNS Record Types

Record Type	Purpose
A Record (Address Record)	Maps a domain to an **IPv4 address**.
AAAA Record	Maps a domain to an **IPv6 address**.
CNAME Record (Canonical Name)	Maps a domain alias to another domain name.
MX Record (Mail Exchange)	Routes emails to a mail server.
TXT Record	Stores text data (used for domain verification & security).

📌 **Example:**
A business maps **www.myapp.com to an EC2 instance** using an **A Record**.

Type: A

Name: www

Value: 192.168.1.1

---

2. Adding an A Record to Point to an EC2 Instance

1. Open **Route 53 Console** → Select **Hosted Zone**.
2. Click **"Create Record"**.
3. **Record Type:** A (Address Record).
4. **Name:** www.myapp.com.
5. **Value:** Enter the **EC2 public IP** (e.g., 54.32.11.100).
6. Click **"Create Record"**.

📌 **Expected Outcome:**
Users accessing www.myapp.com will be directed to your **EC2 instance**.

---

3. Using a CNAME Record for Load Balancer Routing

1. **Record Type:** CNAME.
2. **Name:** www.myapp.com.
3. **Value:** Enter the **ALB DNS Name** (e.g., myapp-alb-123456.us-east-1.elb.amazonaws.com).
4. Click **"Create Record"**.

📌 **Expected Outcome:**
Users accessing www.myapp.com will be routed to the **Load Balancer,** improving scalability.

CHAPTER 4: IMPLEMENTING ROUTE 53 TRAFFIC ROUTING POLICIES

1. Simple Routing (Default Routing)

- Directs all traffic to a **single IP or domain**.
- Example: www.myapp.com → EC2 instance IP.

2. Weighted Routing (Load Distribution)

- Divides traffic across multiple resources based on assigned weights.
- Example: **Send 70% of traffic to AWS ALB and 30% to an on-premises server**.

3. Latency-Based Routing (Global Performance Optimization)

- Routes users to the **nearest AWS region** for better performance.
- Example: Users from **US access us-east-1 while users from Europe access eu-west-1**.

4. Failover Routing (High Availability)

- Redirects traffic to a **secondary resource** when the primary is unhealthy.
- Example: If **primary EC2 instance fails, Route 53 directs traffic to a backup server**.

5. Geolocation Routing (Region-Based Access Control)

- Routes users based on **their geographic location**.
- Example: Users from **India access the India AWS region, while US users access the US AWS region**.

Chapter 5: Configuring Health Checks & Failover

1. Create a Health Check for an EC2 Instance

1. Open **Route 53 Console** → Click **"Health Checks"**.
2. Click **"Create Health Check"**.
3. **Monitor Endpoint:** Enter the **EC2 public IP** or **ALB URL**.
4. **Protocol:** HTTP/HTTPS.
5. **Failure Threshold:** 3 consecutive failures.
6. **Create Health Check** → Attach it to **Failover Routing Policy**.

📌 **Expected Outcome:**
If the **primary server fails,** Route 53 redirects traffic to a **backup server**.

Chapter 6: Route 53 Security Best Practices

✓ Enable Route 53 DNSSEC (Domain Name System Security Extensions).
✓ Restrict changes to Route 53 records using IAM policies.
✓ Use AWS WAF with Route 53 to block malicious traffic.
✓ Monitor DNS query logs using AWS CloudTrail & CloudWatch.
✓ Use TTL (Time-to-Live) values efficiently to control DNS caching.

📌 **Example:**
A **banking website** enables **DNSSEC and Route 53 logging** to **detect suspicious DNS requests**.

Chapter 7: Real-World Use Cases of AWS Route 53

1. E-Commerce Website with Global Traffic

✓ Uses **latency-based routing** to send users to the closest AWS region.
✓ Implements **failover routing** to keep the website available.
✓ Secures DNS records using **AWS WAF & DNSSEC**.

2. API Gateway Routing for Microservices

✓ Uses **subdomains** (e.g., api.myapp.com → AWS API Gateway).
✓ Implements **weighted routing** to test new API versions.
✓ Monitors API health using **Route 53 health checks**.

3. Multi-Region Disaster Recovery Strategy

✓ Uses **failover routing** between AWS US and Europe regions.
✓ Implements **geolocation routing** for region-specific applications.

Conclusion: Mastering AWS Route 53

By using **AWS Route 53**, businesses can:

✅ **Manage DNS records efficiently**.
✅ **Improve website performance with intelligent traffic routing**.
✅ **Implement failover for high availability**.
✅ **Secure DNS queries using AWS security best practices**.

Final Exercise:

1. **Create a hosted zone for a custom domain in Route 53**.
2. **Set up weighted routing between two AWS regions**.
3. **Enable DNSSEC for enhanced security**.

AWS Direct Connect & VPN – Study Material

Introduction to AWS Direct Connect & VPN

What is AWS Direct Connect?

AWS **Direct Connect** is a dedicated **network connection between an on-premises data center and AWS**. It **bypasses the public internet**, providing a more **secure, reliable, and high-speed network link** to AWS services.

What is AWS VPN (Virtual Private Network)?

AWS **VPN** is a **secure, encrypted tunnel** that connects on-premises networks or remote users to AWS. It provides **site-to-site or client-based** connectivity using **IPsec encryption** over the internet.

Chapter 1: AWS Direct Connect vs. AWS VPN – Key Differences

Feature	AWS Direct Connect	AWS VPN
Connection Type	Dedicated private network	Encrypted internet tunnel
Latency & Speed	Low latency (1Gbps – 100Gbps)	Higher latency (depends on ISP)
Security	Private connection (not over the internet)	Encrypted data transmission

Use Case	High-speed AWS access for enterprises	Secure remote access for small to medium businesses
Setup Time	Weeks (physical link required)	Hours (software-based setup)

📌 **Example:**
A **large enterprise with heavy cloud workloads** uses **Direct Connect for high-speed AWS access**, while a **remote development team** uses **AWS VPN for secure remote work**.

CHAPTER 2: AWS DIRECT CONNECT – HOW IT WORKS

1. AWS Direct Connect Components

Component	Description
AWS Direct Connect Location	A physical connection point between AWS and on-premises.
Virtual Interface (VIF)	Logical interface that connects to AWS resources (Private, Public, Transit).
DX Gateway	Connects Direct Connect to multiple VPCs across AWS Regions.
Partner Network	AWS Direct Connect can be provisioned via an **AWS Partner** instead of AWS directly.

---

2. Setting Up AWS Direct Connect

Step 1: Request a Direct Connect Connection

1. Open **AWS Console** → Navigate to **Direct Connect**.
2. Click **"Create Connection"**.
3. **Select AWS Direct Connect Location** (e.g., US-East-1, Equinix DC).
4. **Specify Connection Speed** (e.g., 1 Gbps).
5. Click **"Request Connection"**.

📌 **Expected Outcome:**
AWS **approves the connection request** and provides details for cross-connect setup.

Step 2: Configure the Direct Connect Link

1. Work with your **ISP or data center provider** to set up a **cross-connect** from your router to AWS's router.
2. AWS provides a **LOA-CFA (Letter of Authorization - Connecting Facility Assignment)** for setup.
3. Configure **BGP (Border Gateway Protocol)** on your router to establish communication.

📌 **Expected Outcome:**
A **private network connection** is now established between **your data center and AWS**.

Step 3: Create a Virtual Interface (VIF)

1. Open **AWS Direct Connect Console**.
2. Click **"Create Virtual Interface"**.

3. Choose **VIF Type**:
    - **Private VIF** – Connects to **VPCs via VGW**.
    - **Public VIF** – Connects to **AWS services like S3 & DynamoDB**.
4. Click **"Create VIF"**.

📌 **Expected Outcome:**
Your **Direct Connect link is ready** to send **private traffic to AWS**.

CHAPTER 3: AWS VPN – HOW IT WORKS

1. AWS VPN Types

AWS VPN Type	Description
Site-to-Site VPN	Connects **on-premises networks** to AWS VPC over the internet.
Client VPN	Securely connects **remote users (laptops, mobile devices)** to AWS.

📌 **Example:**
A **company HQ in New York uses Site-to-Site VPN** to connect to AWS, while **remote employees use Client VPN** to work securely.

---

2. Setting Up a Site-to-Site VPN

Step 1: Create a Virtual Private Gateway (VGW)

1. Open **AWS VPC Console** → Click **"Virtual Private Gateways"**.

2. Click **"Create Virtual Private Gateway"**.
3. Enter **VGW Name** (MyVPNGateway).
4. Click **"Create"** → Attach it to your VPC.

📌 **Expected Outcome:**
The **VGW is ready** to handle VPN traffic.

Step 2: Create a Customer Gateway (CGW)

1. Open **AWS VPC Console** → Click **"Customer Gateways"**.
2. Click **"Create Customer Gateway"**.
3. **IP Address:** Enter the **public IP** of your on-premises router.
4. **Routing Type:** Choose **Static (for fixed routes) or Dynamic (for BGP)**.
5. Click **"Create"**.

📌 **Expected Outcome:**
AWS **recognizes your on-premises router** as a VPN peer.

Step 3: Create a Site-to-Site VPN Connection

1. Open **AWS VPC Console** → Click **"VPN Connections"**.
2. Click **"Create VPN Connection"**.
3. Select **Virtual Private Gateway (VGW)** created earlier.
4. Select **Customer Gateway (CGW)** created earlier.
5. Choose **Routing Option** (Static or Dynamic).

6. Click **"Create VPN Connection"**.

📌 **Expected Outcome:**
A **secure VPN tunnel** is established between AWS and your on-premises network.

Step 4: Configure On-Premises Router (IPsec Tunnel)

1. Download the **VPN Configuration File** from AWS.
2. Use **Cisco, Juniper, or Fortinet routers** to establish an **IPsec tunnel**.
3. Configure **IKE Phase 1 & 2 settings** (encryption, authentication).
4. Test the **VPN tunnel status in AWS Console**.

📌 **Expected Outcome:**
The **on-premises router successfully connects to AWS via VPN**.

CHAPTER 4: MONITORING AWS DIRECT CONNECT & VPN

1. Monitoring Direct Connect with CloudWatch

✓ View **network utilization, packet loss, and BGP status**.
✓ Set **CloudWatch Alarms** for network failures.
✓ Use **AWS Direct Connect Resiliency Toolkit** for failover testing.

📌 **Example:**
A **financial company monitors Direct Connect** to ensure **low latency for trading applications**.

---

2. Monitoring AWS VPN Connections

✓ View **VPN connection status in AWS Console**.
✓ Enable **VPC Flow Logs** to track VPN traffic.
✓ Use **AWS Network Manager** for centralized VPN monitoring.

📌 **Example:**
A **global enterprise tracks VPN traffic logs** to **detect unauthorized access attempts**.

Chapter 5: AWS Direct Connect & VPN Best Practices

✓ **Use AWS Direct Connect for low-latency, high-bandwidth workloads**.
✓ **Enable VPN as a backup for Direct Connect** in case of failure.
✓ **Configure BGP routing for dynamic route updates**.
✓ **Use multi-region Direct Connect links** for redundancy.
✓ **Monitor connection health with CloudWatch & VPC Flow Logs**.

📌 **Example:**
A **media streaming company** combines **Direct Connect for high-speed uploads** and **VPN as a backup**.

Chapter 6: Real-World Use Cases of AWS Direct Connect & VPN

1. Hybrid Cloud Deployment

✓ **Direct Connect links on-premises data centers** to AWS for high-performance applications.
✓ **VPN provides secure failover** in case of Direct Connect failures.

2. Remote Workforce Security

✓ **AWS Client VPN allows employees to work remotely** with encrypted access.
✓ **Multi-Factor Authentication (MFA) enhances security**.

3. Disaster Recovery & Business Continuity

✓ **Use Direct Connect for real-time database replication.**
✓ **Enable VPN as a backup for AWS access in emergencies**.

Conclusion: Mastering AWS Direct Connect & VPN

By implementing **AWS Direct Connect & VPN**, businesses can:

✅ **Securely connect on-premises networks to AWS**.
✅ **Reduce latency and improve cloud performance**.
✅ **Ensure failover using VPN as a backup for Direct Connect**.
✅ **Enhance security with encryption and traffic monitoring**.

Final Exercise:

1. **Create a Site-to-Site VPN Connection in AWS**.
2. **Set up Direct Connect using an AWS Partner**.
3. **Monitor VPN connection health using CloudWatch**.

AWS DevOps Tools

AWS CodeCommit, CodeBuild, and CodePipeline – Study Material

Introduction to AWS CI/CD Services

What are AWS CodeCommit, CodeBuild, and CodePipeline?

AWS provides a **fully managed Continuous Integration & Continuous Deployment (CI/CD) pipeline** through its DevOps services:

✓ **AWS CodeCommit** – A **private Git repository** service for source code management.
✓ **AWS CodeBuild** – A **serverless build service** that compiles, tests, and packages applications.
✓ **AWS CodePipeline** – A **CI/CD automation service** that orchestrates build, test, and deployment workflows.

Key Features of AWS CI/CD Services

✓ **Fully managed and scalable** – AWS handles infrastructure and scaling.
✓ **Integration with AWS IAM, S3, EC2, Lambda, and other AWS services**.
✓ **Automated testing & deployment** – Streamlines DevOps processes.
✓ **Supports multiple programming languages (Java, Python, Node.js, etc.)**.
✓ **Security & Compliance** – IAM-based access control and encryption.

CHAPTER 1: AWS CODECOMMIT – SOURCE CODE MANAGEMENT

1. What is AWS CodeCommit?

AWS **CodeCommit** is a **fully managed, secure Git repository** service that enables teams to store and manage their source code in AWS.

2. Benefits of AWS CodeCommit

✓ **Unlimited private repositories** – No size restrictions.
✓ **High availability** – Fully managed, backed by AWS infrastructure.
✓ **Integrated with AWS IAM** – Granular permission control for users.
✓ **Supports Git CLI** – Works with existing Git commands.

---

3. Setting Up AWS CodeCommit

Step 1: Create a CodeCommit Repository

1. Open **AWS CodeCommit Console** → Click **"Create Repository"**.
2. Enter **Repository Name** (MyAppRepo).
3. Add an **optional description**.
4. Click **"Create"**.

Step 2: Connect to the Repository

1. **Clone using HTTPS (For IAM users with AWS credentials)**
2. git clone https://git-codecommit.us-east-1.amazonaws.com/v1/repos/MyAppRepo
3. cd MyAppRepo
4. **Clone using SSH (For SSH key authentication)**

5. git clone ssh://git-codecommit.us-east-1.amazonaws.com/v1/repos/MyAppRepo

Step 3: Push Code to CodeCommit

git add .

git commit -m "Initial commit"

git push origin main

📌 **Expected Outcome:**
Your source code is now stored in **AWS CodeCommit**.

CHAPTER 2: AWS CODEBUILD – AUTOMATED BUILD AND TESTING

1. What is AWS CodeBuild?

AWS **CodeBuild** is a fully managed **build service** that compiles, tests, and packages source code.

2. Benefits of AWS CodeBuild

✓ **Serverless & Scalable** – No need to manage build servers.
✓ **Supports multiple environments** – Works with **Java, Python, Node.js, .NET, Go, etc.**
✓ **Integrates with CodeCommit, GitHub, Bitbucket**.
✓ **Generates real-time logs in Amazon CloudWatch**.

---

3. Setting Up AWS CodeBuild

Step 1: Create a Build Project

1. Open **AWS CodeBuild Console** → Click **"Create Build Project"**.
2. **Enter Project Name:** MyAppBuild.

3. **Source Provider:** Select **AWS CodeCommit** → Choose **MyAppRepo**.
4. **Environment:**
    - Runtime: **Amazon Linux 2**
    - Build Image: **aws/codebuild/standard:5.0**
    - Compute Type: **Small**

Step 2: Define Build Commands in buildspec.yml

Create a buildspec.yml file in the repository root:

```
version: 0.2
phases:
 install:
  runtime-versions:
   nodejs: 14
 build:
  commands:
   - echo "Building Application..."
   - npm install
   - npm run build
artifacts:
 files:
  - '**/*'
```

Step 3: Start the Build

1. Click **"Start Build"**.

2. Monitor build logs in **CloudWatch Logs**.

📌 **Expected Outcome:**
The application code is **compiled and tested successfully**.

CHAPTER 3: AWS CODEPIPELINE – CI/CD AUTOMATION

1. What is AWS CodePipeline?

AWS **CodePipeline** is a CI/CD **orchestration tool** that automates build, test, and deployment workflows.

2. Benefits of AWS CodePipeline

✓ **Automates software release cycles**.
✓ **Integrates with CodeCommit, GitHub, Bitbucket**.
✓ **Supports parallel deployments** for multi-region applications.
✓ **Secure with IAM-based access control**.

---

3. Setting Up AWS CodePipeline

Step 1: Create a New Pipeline

1. Open **AWS CodePipeline Console** → Click **"Create Pipeline"**.
2. **Pipeline Name:** MyAppPipeline.
3. **Select Service Role:** Create new role.
4. Click **"Next"**.

Step 2: Configure Source Stage

1. **Source Provider:** Select **AWS CodeCommit**.
2. Choose **Repository:** MyAppRepo.
3. **Branch Name:** main.

4. Click **"Next"**.

Step 3: Configure Build Stage

1. **Build Provider:** Select **AWS CodeBuild**.
2. Choose **Build Project:** MyAppBuild.
3. Click **"Next"**.

Step 4: Configure Deployment Stage (Example: S3 or Elastic Beanstalk)

1. **Deployment Provider:** Select **AWS S3** or **AWS Elastic Beanstalk**.
2. Choose **S3 Bucket/Environment Name** for deployment.
3. Click **"Next"** → **"Create Pipeline"**.

📌 **Expected Outcome:**
Whenever code is **pushed to CodeCommit,** the pipeline will:
✓ **Automatically build the application using CodeBuild**.
✓ **Deploy the built artifacts to AWS S3 or Elastic Beanstalk**.

CHAPTER 4: MONITORING & TROUBLESHOOTING AWS CI/CD SERVICES

1. Monitoring CodeBuild & CodePipeline

✓ **View real-time logs in AWS CloudWatch**.
✓ **Check build status in the CodeBuild console**.
✓ **Set CloudWatch Alarms for failed builds or deployments**.

2. Troubleshooting Common Issues

✓ **Authentication Issues** – Ensure IAM policies allow CodeBuild and CodePipeline to access S3, CodeCommit, and EC2.

✓ **Build Failures** – Check buildspec.yml for syntax errors.

✓ **Pipeline Execution Failure** – Verify source repository and build artifacts.

Chapter 5: AWS CI/CD Best Practices

✓ **Use CodeCommit for secure source code management.**

✓ **Enable CodeBuild caching for faster builds.**

✓ **Automate deployments using CodePipeline.**

✓ **Use IAM roles to enforce least-privilege access control.**

✓ **Enable CloudWatch monitoring for pipeline visibility.**

📌 **Example:**

A **fintech company** automates code deployment using:

- **CodeCommit for version control**
- **CodeBuild for secure builds**
- **CodePipeline for automated deployment**

Chapter 6: Real-World Use Cases of AWS CodeCommit, CodeBuild & CodePipeline

1. CI/CD for Web Applications

✓ **Developers push code to CodeCommit.**

✓ **CodeBuild compiles and tests the code.**

✓ **CodePipeline deploys to S3 or Elastic Beanstalk.**

2. CI/CD for Serverless Applications (AWS Lambda)

✓ **Build & test Lambda functions using CodeBuild.**

✓ **Deploy to AWS Lambda automatically using CodePipeline.**

3. Automated Security Scanning in CI/CD

✓ Use **CodeBuild with security tools (SonarQube, AWS Inspector)** for **vulnerability scans**.

Conclusion: Mastering AWS CodeCommit, CodeBuild & CodePipeline

By using **AWS CI/CD services**, businesses can:

- ✅ **Automate software builds, testing, and deployments.**
- ✅ **Securely manage source code using CodeCommit.**
- ✅ **Reduce deployment errors with CodePipeline workflows.**

Final Exercise:

1. **Create a CodePipeline that deploys a web application to S3.**
2. **Enable CloudWatch Alarms for failed builds in CodeBuild.**
3. **Implement a Lambda function to notify failures via SNS.**

CI/CD Pipelines with AWS DevOps – Study Material

Introduction to CI/CD in AWS DevOps

What is CI/CD?

CI/CD (**Continuous Integration/Continuous Deployment**) is a DevOps practice that enables **frequent and automated software releases** by integrating code, running tests, and deploying applications seamlessly.

What is AWS DevOps?

AWS provides a suite of **DevOps tools** to automate **software development, testing, deployment, and monitoring**, allowing teams to efficiently implement CI/CD pipelines.

Key AWS CI/CD Tools

✓ **AWS CodeCommit** – Private Git repository for version control.
✓ **AWS CodeBuild** – Automated build and test service.
✓ **AWS CodeDeploy** – Automated deployment service for EC2, Lambda, and ECS.
✓ **AWS CodePipeline** – CI/CD workflow automation tool.
✓ **AWS Elastic Beanstalk** – Simplified application deployment and management.
✓ **AWS CloudFormation** – Infrastructure as Code (IaC) for resource provisioning.

Chapter 1: Understanding CI/CD Pipelines

1. What is a CI/CD Pipeline?

A **CI/CD pipeline** is an **automated workflow** that integrates, tests, and deploys code changes.

2. Phases of a CI/CD Pipeline

Stage	Description	AWS Service Used
Source	Version control for code changes	**AWS CodeCommit, GitHub, Bitbucket**
Build	Compile and test code	**AWS CodeBuild**
Test	Run automated tests	**AWS CodeBuild, AWS Lambda, Selenium**
Deploy	Deploy application to AWS	**AWS CodeDeploy, AWS Elastic Beanstalk, AWS Lambda**
Monitor	Track performance and errors	**Amazon CloudWatch, AWS X-Ray**

📌 **Example:**
A **CI/CD pipeline for a web application** would:

1. **Fetch code from CodeCommit** → 2. **Build and test using CodeBuild** → 3. **Deploy via CodeDeploy to an EC2 instance**.

Chapter 2: Setting Up a CI/CD Pipeline with AWS CodePipeline

1. Step-by-Step Guide to Create a CI/CD Pipeline

Step 1: Create a CodeCommit Repository

1. Open **AWS CodeCommit Console** → Click **"Create Repository"**.

2. Enter **Repository Name:** MyAppRepo.
3. Click **"Create"** → Clone the repo:
4. git clone https://git-codecommit.us-east-1.amazonaws.com/v1/repos/MyAppRepo
5. cd MyAppRepo

Step 2: Create a CodeBuild Project

1. Open **AWS CodeBuild Console** → Click **"Create Build Project"**.
2. **Source:** Choose **CodeCommit** → Select MyAppRepo.
3. **Environment:**
   - Image: aws/codebuild/standard:5.0
   - Runtime: **Amazon Linux 2**
4. **Buildspec file:** Create a buildspec.yml file in the repo:
5. version: 0.2
6. phases:
7. install:
8. runtime-versions:
9. nodejs: 14
10. build:
11. commands:
12. - echo "Building Application..."
13. - npm install
14. - npm run build

15.artifacts:

16. files:

17. - '**/*'

18. Click **"Create Build Project"**.

📌 **Expected Outcome:**
The build service **automates code compilation and testing**.

Step 3: Create an Application Deployment via CodeDeploy

1. Open **AWS CodeDeploy Console** → Click **"Create Application"**.
2. Enter **Application Name:** MyAppDeploy.
3. Choose **Compute Platform:** EC2/On-Premises.
4. Create a **Deployment Group** → Attach an **IAM role**.

📌 **Expected Outcome:**
AWS CodeDeploy is **ready to deploy new versions of the application**.

Step 4: Create a CodePipeline for CI/CD Automation

1. Open **AWS CodePipeline Console** → Click **"Create Pipeline"**.
2. **Pipeline Name:** MyAppPipeline.
3. **Source Provider:** Select **AWS CodeCommit** → Choose MyAppRepo.
4. **Build Provider:** Select **AWS CodeBuild** → Choose MyAppBuild.

5. **Deployment Provider:** Select **AWS CodeDeploy** → Choose MyAppDeploy.
6. Click **"Create Pipeline"**.

📌 **Expected Outcome:**
Whenever **new code is pushed**, AWS **automatically builds, tests, and deploys it**.

CHAPTER 3: AUTOMATING INFRASTRUCTURE DEPLOYMENT WITH AWS CLOUDFORMATION

1. What is AWS CloudFormation?

AWS **CloudFormation** automates the provisioning of AWS resources using **Infrastructure as Code (IaC)**.

2. Creating a CloudFormation Template for CI/CD

1. Create a **CloudFormation template** (cicd.yaml):
2. Resources:
3. MyCodePipeline:
4. Type: AWS::CodePipeline::Pipeline
5. Properties:
6. Name: MyAppPipeline
7. RoleArn: arn:aws:iam::123456789012:role/AWSCodePipelineServiceRole
8. ArtifactStore:
9. Type: S3
10. Location: my-pipeline-artifacts

```
Stages:
        - Name: Source
    Actions:
        - Name: SourceAction
      ActionTypeId:
          Category: Source
       Owner: AWS
          Provider: CodeCommit
          Version: 1
        Configuration:
          RepositoryName: MyAppRepo
          BranchName: main
        OutputArtifacts:
          - Name: SourceArtifact
    Deploy CloudFormation stack:
    aws cloudformation create-stack --stack-name MyCICDPipeline --template-body file://cicd.yaml
```

📌 **Expected Outcome:**
AWS **automatically provisions CI/CD pipelines** using this script.

CHAPTER 4: MONITORING AND LOGGING IN CI/CD PIPELINES

1. AWS CloudWatch for Build Monitoring

✓ **View logs from CodeBuild, CodeDeploy, and EC2 instances**.
✓ **Set up alarms** for failed deployments.

2. AWS X-Ray for Debugging CI/CD Pipelines

✓ **Tracks application requests and errors**.
✓ **Identifies performance bottlenecks**.

📌 **Example:**
A **CodeDeploy failure triggers a CloudWatch Alarm,** alerting developers.

CHAPTER 5: CI/CD BEST PRACTICES WITH AWS DEVOPS

✓ **Use Infrastructure as Code (IaC) with CloudFormation**.
✓ **Enable automated testing in CI/CD pipelines**.
✓ **Enforce IAM least-privilege access for CI/CD users**.
✓ **Use CodeDeploy blue/green deployments for zero-downtime updates**.
✓ **Monitor pipelines with CloudWatch and X-Ray**.

📌 **Example:**
A **fintech company** implements **CI/CD best practices** to **automate software releases while maintaining security**.

CHAPTER 6: REAL-WORLD USE CASES OF CI/CD WITH AWS DEVOPS

1. CI/CD for Microservices (ECS & Lambda)

✓ **Deploy microservices using AWS Fargate & CodePipeline**.
✓ **Use AWS Lambda for serverless automation**.

2. CI/CD for Machine Learning Models

✓ **Retrain models with CodeBuild when new data arrives.**
✓ **Deploy models using AWS SageMaker CI/CD pipelines.**

3. Enterprise DevOps Pipeline

✓ **Multi-region deployments using AWS Global Accelerator.**
✓ **Blue/Green deployments for zero downtime.**

Conclusion: Mastering CI/CD Pipelines with AWS DevOps

By using **AWS DevOps tools,** businesses can:

✅ **Automate code integration, testing, and deployment.**
✅ **Improve release efficiency and software quality.**
✅ **Scale and secure CI/CD pipelines with AWS best practices.**

Final Exercise:

1. **Create a CodePipeline for deploying an S3-hosted static website.**
2. **Set up a CodeDeploy blue/green deployment for an EC2 app.**
3. **Enable CloudWatch alarms for failed build/test steps.**

INFRASTRUCTURE AS CODE (IAC)

INTRODUCTION TO AWS CLOUDFORMATION – STUDY MATERIAL

What is AWS CloudFormation?

AWS **CloudFormation** is an **Infrastructure as Code (IaC) service** that allows users to define and provision AWS resources **automatically using templates**. It helps automate infrastructure management, ensuring consistency, scalability, and repeatability.

CHAPTER 1: UNDERSTANDING AWS CLOUDFORMATION

1. Why Use AWS CloudFormation?

✓ **Infrastructure as Code (IaC)** – Define AWS resources in YAML or JSON templates.
✓ **Automated Resource Management** – Deploy, update, and delete AWS infrastructure easily.
✓ **Scalability & Consistency** – Deploy the same infrastructure across multiple environments.
✓ **Supports Most AWS Services** – EC2, S3, RDS, Lambda, VPC, IAM, and more.
✓ **Version Control & Change Management** – Track infrastructure changes using templates.

📌 **Example:**
A **startup automates AWS resource creation** by defining an EC2 instance, S3 bucket, and security group in a CloudFormation template.

2. Key CloudFormation Concepts

Concept	Description
Stack	A collection of AWS resources created from a CloudFormation template.
Template	A YAML/JSON file defining AWS infrastructure.
StackSet	Allows deployment of stacks across multiple AWS accounts/regions.
Parameters	Customizable input values for templates (e.g., instance type).
Mappings	Static key-value pairs used to define configurations.
Conditions	Define rules to include/exclude resources based on conditions.
Outputs	Returns values after stack creation (e.g., EC2 instance ID).
Change Sets	Shows what will change before applying updates to stacks.

CHAPTER 2: CREATING A CLOUDFORMATION TEMPLATE

1. Basic CloudFormation Template Structure

A CloudFormation template consists of **five main sections**:

AWSTemplateFormatVersion: "2010-09-09"

Description: "Basic CloudFormation Template Example"

Resources:

```
MyS3Bucket:
  Type: "AWS::S3::Bucket"
  Properties:
    BucketName: "my-cloudformation-bucket"
```

✓ **AWSTemplateFormatVersion** – Defines the CloudFormation template version.
✓ **Description** – Provides details about the template.
✓ **Resources** – Defines AWS resources like EC2, S3, RDS, and Lambda.

📌 **Expected Outcome:**
This template **creates an S3 bucket** named my-cloudformation-bucket.

---

2. Defining an EC2 Instance in CloudFormation

```
Resources:
  MyEC2Instance:
    Type: "AWS::EC2::Instance"
    Properties:
      ImageId: "ami-0abcdef1234567890"
      InstanceType: "t2.micro"
      KeyName: "my-key-pair"
      SecurityGroups:
        - Ref: "MySecurityGroup"
```

```
MySecurityGroup:
  Type: "AWS::EC2::SecurityGroup"
  Properties:
    GroupDescription: "Enable SSH Access"
    SecurityGroupIngress:
      - IpProtocol: "tcp"
        FromPort: 22
        ToPort: 22
        CidrIp: "0.0.0.0/0"
```

📌 **Expected Outcome:**

✓ Deploys an **EC2 instance** with a security group allowing SSH access.

CHAPTER 3: DEPLOYING A CLOUDFORMATION STACK

Step 1: Upload Template to CloudFormation

1. Open **AWS Console** → Navigate to **CloudFormation**.
2. Click **"Create Stack"** → **"With New Resources"**.
3. **Choose Template Source:** Upload a **YAML/JSON** template file.
4. Click **"Next"**.

Step 2: Configure Stack Details

1. **Stack Name:** MyCloudFormationStack.
2. Enter **Parameters (if any)**.

3. Click **"Next"** → Configure Stack Options.

Step 3: Review and Deploy Stack

1. Review settings and click **"Create Stack"**.
2. Wait for **Stack Status = CREATE_COMPLETE**.

📌 **Expected Outcome:**

✓ AWS resources (EC2, S3, RDS, etc.) **are created automatically**.

CHAPTER 4: USING PARAMETERS, MAPPINGS, AND OUTPUTS

1. Using Parameters in CloudFormation

Allows users to **customize stack settings** dynamically.

```
Parameters:
  InstanceType:
    Type: String
    Default: "t2.micro"
    AllowedValues:
      - "t2.micro"
      - "t2.small"
      - "t2.medium"
Resources:
  MyEC2Instance:
    Type: "AWS::EC2::Instance"
    Properties:
      InstanceType: !Ref InstanceType
```

📌 **Expected Outcome:**

✓ Users **choose an instance type** when launching the stack.

---

2. Using Mappings for Configurations

Define static key-value pairs.

```
Mappings:
  RegionMap:
    us-east-1:
      AMI: "ami-0abcdef1234567890"
    us-west-1:
      AMI: "ami-0fedcba0987654321"
Resources:
  MyEC2Instance:
    Type: "AWS::EC2::Instance"
    Properties:
      ImageId: !FindInMap [RegionMap, !Ref "AWS::Region", AMI]
```

📌 **Expected Outcome:**

✓ Selects the **AMI based on the AWS region**.

---

3. Using Outputs to Retrieve Stack Details

```
Outputs:
  InstanceID:
    Description: "The ID of the EC2 instance"
```

Value: !Ref MyEC2Instance

📌 **Expected Outcome:**

✓ **Outputs the EC2 instance ID** after deployment.

Chapter 5: Updating and Deleting CloudFormation Stacks

1. Updating a Stack

1. Open **CloudFormation Console** → Select **Stack**.
2. Click **"Update"** → Choose **"Replace Current Template"**.
3. Modify the **YAML/JSON template** → Click **"Update Stack"**.

📌 **Expected Outcome:**

✓ AWS automatically **modifies the existing stack** without downtime.

---

2. Deleting a Stack

1. Open **CloudFormation Console** → Select **Stack**.
2. Click **"Delete Stack"**.
3. Confirm deletion.

📌 **Expected Outcome:**

✓ AWS **removes all resources** created by the stack.

Chapter 6: Monitoring and Troubleshooting CloudFormation

1. Monitoring CloudFormation Stacks

✓ Use **CloudFormation Events** to check stack creation status.
✓ Enable **AWS CloudTrail** to log API calls.
✓ Use **Amazon CloudWatch** for performance monitoring.

2. Common CloudFormation Errors & Solutions

Error	Cause	Solution
ROLLBACK_COMPLETE	Deployment failed	Check logs, update stack with correct settings
IAM Policy Errors	Insufficient permissions	Assign correct IAM roles
Invalid Parameters	Incorrect values in parameters	Validate input values before deployment

---

CHAPTER 7: ADVANCED CLOUDFORMATION FEATURES

1. Nested Stacks

✓ Use **Nested Stacks** to **manage large deployments**.
✓ Break a template into **smaller, reusable stacks**.

📌 **Example:**
A **microservices architecture** with separate CloudFormation templates for **database, backend, and frontend**.

---

2. AWS CloudFormation StackSets

✓ Deploy **CloudFormation stacks across multiple AWS accounts and regions**.

📌 **Example:**
An **enterprise manages multi-region VPCs and IAM roles** using **StackSets**.

Chapter 8: Best Practices for AWS CloudFormation

✓ **Use Version Control** – Store templates in **GitHub, CodeCommit**.
✓ **Implement IAM Least Privilege** – Assign minimal permissions.
✓ **Test in a Dev Environment** – Validate templates before production deployment.
✓ **Enable CloudFormation Drift Detection** – Detect changes made outside CloudFormation.
✓ **Use Parameter Store or Secrets Manager** – Avoid hardcoding credentials.

Conclusion: Mastering AWS CloudFormation

By using **AWS CloudFormation,** businesses can:
✅ **Automate infrastructure provisioning**.
✅ **Deploy scalable, consistent AWS environments**.
✅ **Reduce manual errors with Infrastructure as Code (IaC)**.
✅ **Easily update and manage AWS resources**.

Final Exercise:

1. **Create a CloudFormation template to deploy a VPC, EC2 instance, and RDS database**.
2. **Modify an existing stack using Change Sets**.
3. **Deploy a multi-region architecture using StackSets**

TERRAFORM FOR AWS – STUDY MATERIAL

INTRODUCTION TO TERRAFORM FOR AWS

What is Terraform?

Terraform is an **open-source Infrastructure as Code (IaC) tool** that allows you to **define, provision, and manage AWS infrastructure** using a **declarative configuration language (HCL - HashiCorp Configuration Language)**.

Why Use Terraform for AWS?

✓ **Automates AWS resource provisioning** – No manual setup required.
✓ **Declarative syntax** – Define the desired state, and Terraform will handle the changes.
✓ **Version control** – Store infrastructure configurations in Git for tracking.
✓ **Reusable infrastructure** – Use Terraform modules to deploy standardized environments.
✓ **Multi-cloud support** – Works with AWS, Azure, Google Cloud, and others.

Chapter 1: Key Terraform Concepts

Concept	Description
Providers	APIs that Terraform interacts with (e.g., AWS, Azure, GCP).
Resources	AWS services created using Terraform (e.g., EC2, S3, VPC).
Variables	Custom inputs used in configurations.

State Files	Tracks deployed infrastructure (terraform.tfstate).
Modules	Reusable Terraform configurations.
Outputs	Returns values after Terraform deployment.
Provisioners	Run scripts on AWS instances during creation.

📌 **Example:**

A **Terraform script can deploy an EC2 instance, create an S3 bucket, and configure security groups automatically**.

CHAPTER 2: INSTALLING AND SETTING UP TERRAFORM

1. Install Terraform on Your Machine

For Windows:

1. Download Terraform from Terraform Official Website.
2. Extract and add it to your **System PATH**.
3. Verify installation:
4. terraform -version

For Linux/macOS:

1. Install using Homebrew (macOS):
2. brew install terraform
3. Install using APT (Ubuntu):
4. sudo apt-get update && sudo apt-get install -y terraform
5. Verify installation:
6. terraform -version

📌 **Expected Outcome:** Terraform is successfully installed and ready to use.

CHAPTER 3: WRITING YOUR FIRST TERRAFORM CONFIGURATION FOR AWS

1. Create a Terraform Project Directory

mkdir terraform-aws-demo

cd terraform-aws-demo

2. Define AWS Provider and Resources

Create a file named main.tf with the following configuration:

```
provider "aws" {
 region = "us-east-1"
}

resource "aws_instance" "my_ec2" {
 ami           = "ami-0abcdef1234567890" # Change to a valid AMI
 instance_type = "t2.micro"

 tags = {
  Name = "Terraform-EC2"
 }
}
```

📌 **Explanation:**

✓ Defines **AWS as the provider**.

✓ Creates an **EC2 instance** with **t2.micro instance type**.

✓ Assigns the **tag "Terraform-EC2"** to the instance.

---

3. Initialize Terraform

Run the following command to initialize Terraform:

terraform init

📌 **Expected Outcome:** Downloads AWS provider plugins and initializes Terraform.

---

4. Plan Infrastructure Deployment

Run the following command to preview what Terraform will create:

terraform plan

📌 **Expected Outcome:** Displays **planned AWS resources** without creating them.

---

5. Apply Terraform Configuration

To create AWS resources, run:

terraform apply

✓ Type **"yes"** when prompted to confirm deployment.

📌 **Expected Outcome:** AWS EC2 instance is successfully created.

6. Destroy Infrastructure (Cleanup)

To remove all AWS resources created by Terraform:

terraform destroy

✓ Type **"yes"** when prompted.

📌 **Expected Outcome:** Terraform removes all AWS resources it created.

CHAPTER 4: USING VARIABLES IN TERRAFORM

1. Define Variables

Modify main.tf to use **variables** instead of hardcoded values:

```
variable "aws_region" {
  default = "us-east-1"
}

variable "instance_type" {
  default = "t2.micro"
}

provider "aws" {
  region = var.aws_region
}
```

```
resource "aws_instance" "my_ec2" {
 ami           = "ami-0abcdef1234567890"
 instance_type = var.instance_type

 tags = {
  Name = "Terraform-EC2"
 }
}
```

2. Override Variables at Runtime

To specify variables dynamically:

```
terraform apply -var="instance_type=t3.micro"
```

📌 **Expected Outcome:**
Terraform deploys an **EC2 instance with a t3.micro type** instead of the default t2.micro.

CHAPTER 5: TERRAFORM STATE MANAGEMENT

1. What is Terraform State?

Terraform maintains the **state of the infrastructure** in a file called terraform.tfstate. This helps track changes over time.

2. Storing State in S3 for Collaboration

Modify main.tf to store state in an S3 bucket:

```
terraform {
 backend "s3" {
```

```
    bucket = "my-terraform-state-bucket"
    key    = "terraform.tfstate"
    region = "us-east-1"
  }
}
```

📌 **Expected Outcome:**
Terraform will **store the state file in S3,** allowing multiple users to collaborate.

CHAPTER 6: USING TERRAFORM MODULES FOR REUSABILITY

1. What Are Terraform Modules?

Terraform **modules** allow you to **reuse infrastructure configurations**.

2. Creating a Simple Terraform Module

Create a folder structure:

```
mkdir modules
mkdir modules/ec2_instance
```

Inside modules/ec2_instance/main.tf, define an EC2 module:

```
variable "instance_type" {}
variable "ami" {}

resource "aws_instance" "my_ec2" {
  ami           = var.ami
```

```
  instance_type = var.instance_type
}
```

Now, call this module in main.tf:

```
module "ec2_instance" {
  source       = "./modules/ec2_instance"
  instance_type = "t2.micro"
  ami          = "ami-0abcdef1234567890"
}
```

📌 **Expected Outcome:**
Terraform **reuses the module** to deploy multiple instances.

CHAPTER 7: MANAGING TERRAFORM WORKSPACES

1. What Are Workspaces?

Terraform **workspaces** allow managing **multiple environments** (e.g., Dev, Staging, Production) with the same codebase.

2. Creating Workspaces

1. **Create a New Workspace**
2. terraform workspace new dev
3. **Switch to a Different Workspace**
4. terraform workspace select production
5. **List All Workspaces**
6. terraform workspace list

📌 **Expected Outcome:**
Terraform manages **separate environments** using workspaces.

Chapter 8: Best Practices for Terraform in AWS

✓ **Use Terraform Cloud or S3 Backend** to manage state remotely.
✓ **Implement Terraform Modules** to reuse configurations.
✓ **Use Terraform Variables & Outputs** for flexible deployment.
✓ **Use Terraform Workspaces** for multiple environments.
✓ **Enable IAM Least Privilege** to restrict Terraform access.

📌 **Example:**
A **DevOps team uses Terraform Cloud** to **collaborate and manage AWS infrastructure** securely.

Chapter 9: Real-World Use Cases of Terraform for AWS

1. Multi-Cloud Deployments

✓ Deploy AWS & Azure resources **from the same Terraform configuration**.

2. Auto Scaling & Load Balancers

✓ Use Terraform to create an **AWS ALB with EC2 Auto Scaling**.

3. Secure Cloud Environments

✓ Automate IAM policies, security groups, and VPC configurations.

4. Disaster Recovery & Backups

✓ Use Terraform **to spin up AWS resources in case of failure**.

Conclusion: Mastering Terraform for AWS

By using **Terraform for AWS**, businesses can:

- ✅ **Automate AWS infrastructure provisioning**.
- ✅ **Ensure consistent deployments** across environments.
- ✅ **Manage AWS at scale with infrastructure as code (IaC)**.

Final Exercise:

1. **Create a Terraform module to deploy an S3 bucket with encryption.**
2. **Use Terraform Workspaces to manage Dev, Staging, and Production environments.**
3. **Store Terraform state in S3 and enable locking with DynamoDB.**

ASSIGNMENT

CONFIGURE A SECURE VPC WITH SUBNETS

SOLUTION: CONFIGURE A SECURE VPC WITH SUBNETS (STEP-BY-STEP GUIDE)

This step-by-step guide will walk you through **configuring a secure AWS VPC with subnets,** including **public and private subnets, security groups, network ACLs,** and **internet access controls**.

Step 1: Log in to AWS and Navigate to VPC Console

1. Open the **AWS Management Console**.
2. Search for **"VPC"** and click on **VPC Dashboard**.
3. Click **"Create VPC"**.

Step 2: Create a VPC

1. **VPC Name:** Secure-VPC
2. **IPv4 CIDR Block:** 10.0.0.0/16 (Allows 65,536 IP addresses)
3. **IPv6 CIDR Block:** No IPv6 for now (optional).
4. **Tenancy:** Default
5. Click **"Create VPC"**.

📌 **Expected Outcome:**
A **VPC is created** with a private network range of 10.0.0.0/16.

Step 3: Create Public and Private Subnets

Create Public Subnet

1. Go to **VPC Console** → **Subnets** → Click **"Create Subnet"**.

2. **Subnet Name:** Public-Subnet-1.
3. **VPC:** Select Secure-VPC.
4. **Availability Zone:** Choose us-east-1a (or any AZ).
5. **IPv4 CIDR Block:** 10.0.1.0/24 (256 IP addresses).
6. Click **"Create Subnet"**.

Create Private Subnet

1. Click **"Create Subnet"** again.
2. **Subnet Name:** Private-Subnet-1.
3. **VPC:** Select Secure-VPC.
4. **Availability Zone:** Choose us-east-1b.
5. **IPv4 CIDR Block:** 10.0.2.0/24.
6. Click **"Create Subnet"**.

📌 **Expected Outcome:**
✓ A **public subnet (10.0.1.0/24)** for internet-facing resources.
✓ A **private subnet (10.0.2.0/24)** for internal applications (no internet access).

Step 4: Attach an Internet Gateway (IGW) for Public Subnet

1. Go to **VPC Console** → **Internet Gateways** → Click **"Create Internet Gateway"**.
2. **Name:** Secure-IGW.
3. Click **"Create"**.
4. **Attach IGW to Secure-VPC:**

- Select **Secure-IGW** → Click **"Actions"** → **"Attach to VPC"**.
- Choose Secure-VPC and click **"Attach"**.

📌 **Expected Outcome:**
The **public subnet can connect to the internet** through the **Internet Gateway**.

Step 5: Configure Route Tables

Create a Public Route Table

1. Go to **VPC Console** → **Route Tables** → Click **"Create Route Table"**.
2. **Name:** Public-RT.
3. **VPC:** Select Secure-VPC.
4. Click **"Create"**.

Add Route for Internet Access

1. Select Public-RT → Click **"Edit Routes"**.
2. Click **"Add Route"**.
3. **Destination:** 0.0.0.0/0 (All Internet Traffic).
4. **Target:** Select **Secure-IGW** (Internet Gateway).
5. Click **"Save Changes"**.

Associate Public Route Table with Public Subnet

1. Click **"Subnet Associations"**.
2. Click **"Edit Subnet Associations"**.
3. Select Public-Subnet-1 → Click **"Save"**.

📌 **Expected Outcome:**

✓ The **public subnet (10.0.1.0/24) now has internet access** via the **Internet Gateway**.

Step 6: Configure a NAT Gateway for Private Subnet (Optional)

A **NAT Gateway** allows private subnet instances to **access the internet** (for updates, etc.) without being exposed to incoming internet traffic.

1. Go to **VPC Console** → **NAT Gateways** → Click **"Create NAT Gateway"**.
2. **Subnet:** Select Public-Subnet-1.
3. **Elastic IP:** Click **"Allocate New Elastic IP"**.
4. Click **"Create NAT Gateway"**.

Add Route for NAT Gateway in Private Route Table

1. Go to **VPC Console** → **Route Tables** → Click **"Create Route Table"**.
2. **Name:** Private-RT.
3. **VPC:** Secure-VPC.
4. Click **"Create"**.

Add NAT Gateway Route

1. Select Private-RT → Click **"Edit Routes"**.
2. Click **"Add Route"**.
3. **Destination:** 0.0.0.0/0.
4. **Target:** Select **NAT Gateway**.

5. Click **"Save Changes"**.

Associate Private Route Table with Private Subnet

1. Click **"Subnet Associations"** → Click **"Edit Subnet Associations"**.
2. Select Private-Subnet-1 → Click **"Save"**.

📌 **Expected Outcome:**

✓ **Instances in the private subnet can access the internet** (for software updates) but **cannot receive incoming traffic**.

Step 7: Configure Security Groups

1. Create a Security Group for Public EC2 Instances

1. Go to **VPC Console** → **Security Groups** → Click **"Create Security Group"**.
2. **Name:** Public-SG.
3. **VPC:** Secure-VPC.
4. **Inbound Rules:**
   - Allow HTTP (80) from 0.0.0.0/0.
   - Allow HTTPS (443) from 0.0.0.0/0.
   - Allow SSH (22) only from **your IP** (your-public-ip/32).
5. Click **"Create Security Group"**.

📌 **Expected Outcome:**

✓ **Public EC2 instances can receive web traffic but are secured against unauthorized access.**

---

2. Create a Security Group for Private EC2 Instances

1. **Name:** Private-SG.
2. **VPC:** Secure-VPC.
3. **Inbound Rules:**
    - Allow MySQL (3306) from **Public-SG** (for database access).
    - Allow internal SSH access (22) only from Public-SG.
4. Click **"Create Security Group"**.

📌 **Expected Outcome:**
✓ Private EC2 instances (e.g., databases) are only accessible from public instances.

Step 8: Configure Network ACLs (Optional, for Additional Security)

1. Go to **VPC Console** → **Network ACLs** → Click **"Create Network ACL"**.
2. **Name:** Secure-NACL.
3. **VPC:** Secure-VPC.
4. Click **"Create"**.

Edit Inbound Rules:

1. Allow HTTP (80), HTTPS (443), and SSH (22) only from **trusted sources**.
2. Deny all other inbound traffic.

📌 **Expected Outcome:**

✓ **Provides an extra layer of security to prevent unauthorized traffic.**

Step 9: Launch and Test an EC2 Instance in Public Subnet

1. Go to **EC2 Console** → Click **"Launch Instance"**.
2. Choose **Amazon Linux 2** or **Ubuntu** AMI.
3. Select **Instance Type:** t2.micro.
4. Choose **Network:** Secure-VPC.
5. Choose **Subnet:** Public-Subnet-1.
6. Select **Security Group:** Public-SG.
7. Click **"Launch"**.

Connect to EC2 via SSH

ssh -i my-key.pem ec2-user@your-public-ip

📌 **Expected Outcome:**

✓ Successfully connected to the EC2 instance in the **public subnet**.

Conclusion: Successfully Configured a Secure VPC with Subnets

✅ **Created a VPC with public and private subnets.**
✅ **Configured an Internet Gateway for public access.**
✅ **Set up a NAT Gateway for private subnet internet access.**
✅ **Applied Security Groups and Network ACLs for secure networking.**

FINAL EXERCISE:

1. **Launch an EC2 instance in the private subnet and test internet access through NAT Gateway.**
2. **Deploy an RDS instance in the private subnet with security group restrictions.**
3. **Enable AWS VPC Flow Logs for monitoring network traffic.**

DEPLOY AN APPLICATION USING AWS CODEPIPELINE

SOLUTION: DEPLOY AN APPLICATION USING AWS CODEPIPELINE (STEP-BY-STEP GUIDE)

This guide will walk you through **deploying an application using AWS CodePipeline**, integrating **CodeCommit (source), CodeBuild (build), and CodeDeploy (deployment)** for **continuous integration and deployment (CI/CD)**.

Step 1: Set Up an AWS CodeCommit Repository

AWS **CodeCommit** is a **Git-based repository** used for storing application source code.

1. Create a CodeCommit Repository

1. Open the **AWS CodeCommit Console**.
2. Click **"Create Repository"**.
3. **Repository Name:** MyAppRepo.
4. Click **"Create"**.

2. Clone the Repository Locally

1. Copy the repository **clone URL** from CodeCommit.
2. Open the terminal and run:
3. git clone https://git-codecommit.us-east-1.amazonaws.com/v1/repos/MyAppRepo
4. cd MyAppRepo

3. Add Sample Application Code

1. Create an **index.html** file:

2. <html>
3. <body>
4. <h1>Deployed via AWS CodePipeline!</h1>
5. </body>
6. </html>
7. Commit and push the changes:
8. git add .
9. git commit -m "Initial commit"
10. git push origin main

📌 **Expected Outcome:**
Your **source code is now stored in AWS CodeCommit**.

Step 2: Create an AWS CodeBuild Project

AWS **CodeBuild** compiles and packages your application.

1. Create a buildspec.yml File

Inside the repository (MyAppRepo), create buildspec.yml:

```
version: 0.2
phases:
 install:
  runtime-versions:
   nodejs: 14
 build:
  commands:
```

```
    - echo "Building application..."
    - mkdir artifacts
    - cp index.html artifacts/
artifacts:
 files:
  - artifacts/index.html
```

2. Configure CodeBuild in AWS Console

1. Open **AWS CodeBuild Console** → Click **"Create Build Project"**.
2. **Project Name:** MyAppBuild.
3. **Source Provider:** Select **AWS CodeCommit,** then select MyAppRepo.
4. **Environment:**
   - Operating System: **Amazon Linux 2**
   - Runtime: **Standard**
   - Image: aws/codebuild/standard:5.0
   - Compute: **Small**
5. **Buildspec File:** Choose **"Use a buildspec file"**.
6. Click **"Create Build Project"**.

📌 **Expected Outcome:**
CodeBuild **automates application build and artifact creation**.

Step 3: Set Up AWS CodeDeploy

AWS **CodeDeploy** automates deploying applications to **EC2, ECS, or Lambda**.

1. Launch an EC2 Instance for Deployment

1. Open **AWS EC2 Console** → Click **"Launch Instance"**.
2. Choose **Amazon Linux 2** or **Ubuntu** AMI.
3. Select **Instance Type:** t2.micro.
4. **Create or Select Key Pair** (to SSH into the instance).
5. **Network Settings:**
    - Choose **VPC and Public Subnet**.
    - Enable **Auto-Assign Public IP**.
6. **Add Storage:** Default (8GB).
7. **Configure Security Group:**
    - Allow HTTP (80), HTTPS (443), and SSH (22).
8. Click **"Launch"**.

2. Install CodeDeploy Agent on EC2

SSH into the EC2 instance:

```
ssh -i my-key.pem ec2-user@your-ec2-public-ip
```

Install the **AWS CodeDeploy agent**:

```
sudo yum update -y
sudo yum install ruby -y
sudo yum install wget -y
cd /home/ec2-user
```

wget https://aws-codedeploy-us-east-1.s3.amazonaws.com/latest/install

chmod +x ./install

sudo ./install auto

sudo systemctl start codedeploy-agent

sudo systemctl enable codedeploy-agent

3. Create an IAM Role for CodeDeploy

1. Open **IAM Console** → Click **"Roles"** → **"Create Role"**.
2. Select **EC2** as a trusted entity.
3. Attach the **AWSCodeDeployRole** policy.
4. Name the role **CodeDeployEC2Role** → Click **"Create"**.
5. Assign this role to the **EC2 instance**.

📌 **Expected Outcome:**
EC2 instance is **ready to receive deployments from AWS CodeDeploy**.

---

4. Create an AWS CodeDeploy Application

1. Open **AWS CodeDeploy Console** → Click **"Create Application"**.
2. **Application Name:** MyAppDeploy.
3. **Compute Platform:** EC2/On-Premises.
4. Click **"Create Application"**.

5. Create a Deployment Group

1. Open the newly created **MyAppDeploy** application.

2. Click **"Create Deployment Group"**.
3. **Deployment Group Name:** MyAppDeploymentGroup.
4. **Service Role:** Select **AWSCodeDeployRole**.
5. **Environment Configuration:**
    - Select **EC2 Instances**.
    - Choose instances tagged with CodeDeployInstance.
6. Click **"Create Deployment Group"**.

📌 **Expected Outcome:**
CodeDeploy is ready to **automate deployments to EC2 instances**.

Step 4: Create AWS CodePipeline

AWS **CodePipeline** automates the CI/CD workflow by integrating **CodeCommit, CodeBuild, and CodeDeploy**.

1. Create a New Pipeline

1. Open **AWS CodePipeline Console** → Click **"Create Pipeline"**.
2. **Pipeline Name:** MyAppPipeline.
3. **Service Role:** Choose **"Create New Role"**.
4. Click **"Next"**.

2. Configure Source Stage

1. **Source Provider:** Select **AWS CodeCommit**.
2. Choose **Repository:** MyAppRepo.
3. **Branch:** main.
4. Click **"Next"**.

3. Configure Build Stage

1. **Build Provider:** Select **AWS CodeBuild**.
2. Choose **Build Project:** MyAppBuild.
3. Click **"Next"**.

4. Configure Deploy Stage

1. **Deploy Provider:** Select **AWS CodeDeploy**.
2. Choose **Application Name:** MyAppDeploy.
3. Choose **Deployment Group:** MyAppDeploymentGroup.
4. Click **"Next"** → **"Create Pipeline"**.

📌 **Expected Outcome:**
CodePipeline **automates deployments from CodeCommit to EC2 via CodeDeploy**.

Step 5: Trigger the CI/CD Pipeline

1. Make a Code Change and Push to CodeCommit

```
echo "<h1>Updated Deployment via AWS CodePipeline!</h1>" > index.html

git add .

git commit -m "Updated index.html"

git push origin main
```

2. Monitor Pipeline Execution

1. Open **AWS CodePipeline Console**.
2. Click on **MyAppPipeline**.

3. Monitor **Source, Build, and Deploy Stages**.

📌 **Expected Outcome:**

✓ The pipeline automatically detects the code change, builds the application, and **deploys it to EC2**.

Step 6: Verify Deployment on EC2 Instance

1. Open **AWS EC2 Console** → Select your instance.
2. Click **"Public IP Address"** to open the app in a browser.

📌 **Expected Outcome:**

Your **updated application is deployed via AWS CodePipeline**! 🎉

Conclusion: Successfully Deployed an Application Using AWS CodePipeline

✅ **Created a CodeCommit repository for version control.**

✅ **Configured CodeBuild to compile and package the application.**

✅ **Deployed the application to EC2 using CodeDeploy.**

✅ **Automated the CI/CD pipeline using AWS CodePipeline.**

Final Exercise:

1. **Modify buildspec.yml to include testing before deployment.**
2. **Use AWS Lambda as a deployment target instead of EC2.**
3. **Integrate Amazon S3 as a deployment destination for static websites.**

WRITE AN AWS CLOUDFORMATION TEMPLATE

SOLUTION: WRITE AN AWS CLOUDFORMATION TEMPLATE (STEP-BY-STEP GUIDE)

This guide will walk you through **writing an AWS CloudFormation template** to deploy **an EC2 instance, an S3 bucket, and a security group**.

Step 1: Understand AWS CloudFormation

What is AWS CloudFormation?

AWS CloudFormation allows you to **define infrastructure as code (IaC)** using YAML or JSON templates. It automates **resource creation, updates, and management**.

Step 2: Install and Set Up AWS CLI (Optional)

You can deploy CloudFormation templates using the AWS Console or AWS CLI.

1. **Install AWS CLI** (Download Here).
2. Configure AWS CLI:
3. aws configure

Enter AWS Access Key, Secret Key, Region (e.g., us-east-1), and output format (json).

Step 3: Write the CloudFormation Template (YAML Format)

1. Create a YAML File for CloudFormation

Create a file named cloudformation-template.yaml.

2. Define the Template Structure

Every CloudFormation template has these key sections:

AWSTemplateFormatVersion: "2010-09-09"

Description: "AWS CloudFormation Template to deploy EC2, S3, and Security Group"

```
Resources:
 MyS3Bucket:
  Type: "AWS::S3::Bucket"
  Properties:
   BucketName: "my-cloudformation-bucket-12345"

 MySecurityGroup:
  Type: "AWS::EC2::SecurityGroup"
  Properties:
   GroupDescription: "Allow SSH and HTTP access"
   SecurityGroupIngress:
    - IpProtocol: "tcp"
      FromPort: 22
      ToPort: 22
      CidrIp: "0.0.0.0/0"
    - IpProtocol: "tcp"
```

```
      FromPort: 80
      ToPort: 80
      CidrIp: "0.0.0.0/0"

  MyEC2Instance:
    Type: "AWS::EC2::Instance"
    Properties:
      InstanceType: "t2.micro"
      ImageId: "ami-0abcdef1234567890"  # Replace with a valid AMI ID
      SecurityGroups:
        - Ref: "MySecurityGroup"
      KeyName: "my-key-pair"  # Replace with an existing key pair

Outputs:
  EC2InstancePublicIP:
    Description: "Public IP of the EC2 Instance"
    Value: !GetAtt MyEC2Instance.PublicIp
  S3BucketName:
    Description: "Name of the created S3 bucket"
    Value: !Ref MyS3Bucket
```

Step 4: Validate the Template

Run the following command to **validate** the YAML syntax:

aws cloudformation validate-template --template-body file://cloudformation-template.yaml

📌 **Expected Outcome:**
If the template is correct, AWS will **return a success message**.

Step 5: Deploy the CloudFormation Stack

1. Using AWS Console

1. Open **AWS Console** → Navigate to **CloudFormation**.
2. Click **"Create Stack"** → Select **"With new resources"**.
3. Choose **"Upload a template file"**, then select cloudformation-template.yaml.
4. Click **"Next"** and enter:
    - **Stack Name:** MyCloudFormationStack.
5. Click **"Next"** → **"Create Stack"**.

2. Using AWS CLI

Run the following command:

aws cloudformation create-stack --stack-name MyCloudFormationStack --template-body file://cloudformation-template.yaml

📌 **Expected Outcome:**
AWS will **provision the EC2 instance, S3 bucket, and security group**.

Step 6: Verify the Stack Deployment

1. Using AWS Console

1. Open **AWS CloudFormation Console**.
2. Click on **MyCloudFormationStack**.
3. Check the **"Resources"** tab to see deployed resources.
4. Go to **EC2 Console** → Find the **Instance ID and Public IP**.
5. Go to **S3 Console** → Find the **created S3 bucket**.

2. Using AWS CLI

To check the stack status:

```
aws cloudformation describe-stacks --stack-name MyCloudFormationStack
```

To find the EC2 public IP:

```
aws ec2 describe-instances --query 'Reservations[*].Instances[*].PublicIpAddress' --output text
```

📌 **Expected Outcome:**
✓ An **EC2 instance** is running with SSH & HTTP access.
✓ An **S3 bucket** is created.

Step 7: Update the CloudFormation Stack

To modify resources, edit cloudformation-template.yaml, then run:

```
aws cloudformation update-stack --stack-name MyCloudFormationStack --template-body file://cloudformation-template.yaml
```

📌 **Example Update:**
Change InstanceType from t2.micro to t3.micro.

Step 8: Delete the CloudFormation Stack

To **clean up resources,** run:

```
aws cloudformation delete-stack --stack-name MyCloudFormationStack
```

📌 **Expected Outcome:**
✓ AWS **removes all resources** created by the stack.

Conclusion: Successfully Created an AWS CloudFormation Template

- ✅ **Defined a YAML CloudFormation template.**
- ✅ **Deployed an EC2 instance, S3 bucket, and Security Group.**
- ✅ **Validated and updated the CloudFormation stack.**
- ✅ **Deleted the stack to clean up resources.**

Final Exercise:

1. **Modify the template to create an RDS database in a private subnet.**
2. **Use CloudFormation Outputs to return the instance's private IP.**
3. **Enable automatic stack rollback on failure.**