Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

INTRODUCTION TO DEVOPS WITH LINUX

CHAPTER 1: UNDERSTANDING DEVOPS AND ITS IMPORTANCE

What is DevOps?

DevOps is a **software development methodology** that integrates **development (Dev)** and **operations (Ops)** teams to improve **collaboration, automation, and continuous delivery** of applications. It eliminates traditional silos between developers and IT operations, leading to **faster deployments, higher efficiency, and improved system reliability**.

The primary goals of DevOps include:

- **Automating software development processes** using tools such as Git, Jenkins, and Ansible.
- **Improving collaboration** between development, testing, and operations teams.
- **Ensuring continuous integration (CI) and continuous deployment (CD)** to deliver software faster.
- **Enhancing monitoring and security** to detect and prevent issues before they affect production systems.

DevOps is crucial for organizations that rely on **rapid software releases**, cloud computing, and containerized applications. By implementing DevOps best practices, companies can **reduce**

ISDM-NxT

software development cycles, improve scalability, and increase system reliability.

CHAPTER 2: ROLE OF LINUX IN DEVOPS

Why Linux is Essential for DevOps

Linux is the **foundation of modern DevOps** because of its **stability, security, flexibility, and compatibility** with automation tools. Most **cloud computing environments, containerized applications, and CI/CD pipelines** run on Linux-based systems.

Key reasons Linux is widely used in DevOps:

- **Open-source nature** – Linux is free and customizable.
- **Command-line interface (CLI) efficiency** – Linux provides powerful scripting and automation capabilities.
- **Server and cloud compatibility** – Most cloud providers (AWS, Google Cloud, Azure) use Linux as their primary OS.
- **Containerization support** – Technologies like Docker and Kubernetes are built for Linux environments.
- **Security and access control** – Linux provides robust security features such as **SELinux, firewalls, and role-based access control (RBAC)**.

In a DevOps workflow, Linux is used for **automating server management, managing cloud infrastructure, deploying applications, and configuring security policies**.

Example: In a **CI/CD pipeline**, a **Linux-based Jenkins server** can automate the testing and deployment of a web application to **AWS EC2 instances running Ubuntu**.

ISDM-NxT

CHAPTER 3: KEY DEVOPS TOOLS IN LINUX

1. Version Control: Git

Git is a distributed **version control system (VCS)** used in DevOps for **tracking changes, collaboration, and managing source code repositories**.

- **Install Git on Linux:**
- sudo apt install git # Debian/Ubuntu
- sudo yum install git # CentOS/RHEL
- **Basic Git Commands:**
- git init # Initialize a new repository
- git clone <repo> # Clone an existing repository
- git commit -m "Message" # Save changes locally
- git push origin main # Push changes to remote repository

2. Continuous Integration & Deployment (CI/CD): Jenkins

Jenkins is an open-source **CI/CD automation tool** that allows developers to **automate build, test, and deployment** processes.

- **Install Jenkins on Linux:**
- sudo apt update
- sudo apt install openjdk-11-jdk
- sudo apt install jenkins
- sudo systemctl enable --now jenkins

3. Configuration Management: Ansible

Ansible automates **server provisioning, configuration, and application deployment**.

- **Install Ansible:**
- sudo apt install ansible # Debian/Ubuntu
- sudo yum install ansible # CentOS/RHEL
- **Run an Ansible Playbook:**
- ansible-playbook site.yml

4. Containerization & Orchestration: Docker & Kubernetes

Containers help **package applications with their dependencies** for consistency across environments.

- **Install Docker on Linux:**
- sudo apt install docker.io
- sudo systemctl enable --now docker
- **Run a container:**
- docker run -d -p 80:80 nginx

Kubernetes is used for **orchestrating multiple containers** in production.

Chapter 4: Automating DevOps Workflows with Linux Scripting

Using Bash Scripts for Automation

ISDM-NXT

Bash scripting is an essential skill for DevOps engineers to **automate server configurations, deploy applications, and manage cloud infrastructure**.

Example: Automating Web Server Deployment

#!/bin/bash

sudo apt update

sudo apt install -y apache2

sudo systemctl start apache2

echo "Web Server Deployed Successfully"

Save the script as deploy.sh, then execute:

chmod +x deploy.sh

./deploy.sh

Using Cron Jobs for Scheduling Tasks

Cron jobs allow scheduling **automated backups, system monitoring, and cleanup tasks**.

To create a cron job that runs every night at midnight:

crontab -e

Add the following line:

0 0 * * * /usr/bin/backup.sh

Chapter 5: Case Study – Implementing DevOps for a Web Application

ISDM-NXT

Scenario:

A startup wants to **automate the deployment of a Python web application** to AWS servers using DevOps tools.

Solution Using Linux DevOps Workflow:

1. **Code Version Control:** Developers push changes to **GitHub**.
2. **CI/CD Pipeline:** Jenkins pulls code from GitHub and runs tests.
3. **Containerization:** Docker packages the application into a container.
4. **Configuration Management:** Ansible automates provisioning of AWS servers.
5. **Orchestration:** Kubernetes manages multiple containers in production.

Outcome:

- **Application deployments are fully automated.**
- **Developers can release new features faster.**
- **System downtime is minimized through automated rollbacks.**

ISDM-NXT

Chapter 6: Exercise

1. **Set up a Git repository on a Linux machine and push a sample project to GitHub.**
2. **Install and configure Jenkins to automate a build process.**

3. **Write a Bash script to automate the installation of Apache and MySQL on a Linux server.**
4. **Deploy a Docker container running Nginx and expose it on port 80.**
5. **Use Ansible to configure a Linux server with predefined roles.**

CONCLUSION

DevOps and Linux go **hand in hand** to streamline software development, **increase automation**, and **enhance system reliability**. By mastering **Linux-based DevOps tools** like Git, Jenkins, Ansible, Docker, and Kubernetes, engineers can **build highly efficient, automated, and scalable IT environments**.

ISDM-NxT

DOCKER & CONTAINERIZATION

CHAPTER 1: INTRODUCTION TO DOCKER AND CONTAINERIZATION

What is Containerization?

Containerization is a **lightweight virtualization technique** that allows applications to be packaged with **all dependencies, libraries, and configurations** into a single unit called a **container**. This ensures that applications **run consistently across different environments**, from development to production.

Why is Containerization Important?

- **Eliminates "works on my machine" issues** by ensuring environment consistency.
- **Enhances scalability and efficiency** in cloud-native applications.
- **Speeds up application deployment** through lightweight, portable containers.
- **Improves security** by isolating applications from the host system.

What is Docker?

Docker is the **most popular containerization platform**, enabling developers to **build, ship, and run** applications inside containers. Unlike traditional virtual machines (**VMs**), Docker containers **share the host OS kernel**, making them **lighter, faster, and more resource-efficient**.

CHAPTER 2: DOCKER VS. VIRTUAL MACHINES (VMS)

ISDM-NxT

Key Differences Between Docker and VMs

Feature	Docker (Containers)	Virtual Machines (VMs)
Performance	Faster startup, lightweight	Slower due to full OS overhead
Resource Usage	Shares host OS kernel	Requires separate OS per VM
Portability	Highly portable	Limited portability
Isolation	Process-level isolation	Full OS isolation
Boot Time	Seconds	Minutes

Example Scenario

A **developer working on a web application** needs a consistent environment across multiple team members. Instead of setting up **multiple VMs**, Docker allows them to **create a container with the required libraries, which can be shared easily across all team members**.

Chapter 3: Installing Docker on Linux

1. Installing Docker on Debian/Ubuntu

sudo apt update

sudo apt install -y docker.io

sudo systemctl enable --now docker

2. Installing Docker on CentOS/RHEL

sudo yum install -y docker

ISDM-NXT

sudo systemctl enable --now docker

3. Verifying Docker Installation

docker --version

To test Docker:

docker run hello-world

This command pulls a test image from **Docker Hub** and runs it.

CHAPTER 4: BASIC DOCKER COMMANDS

1. Running a Docker Container

To run a basic **Nginx web server container**:

docker run -d -p 80:80 nginx

- -d → Runs the container in detached mode (background).
- -p 80:80 → Maps port 80 of the host to port 80 of the container.

2. Listing Running Containers

docker ps

To list **all containers (including stopped ones)**:

docker ps -a

3. Stopping and Removing Containers

To stop a running container:

docker stop <container_id>

ISDM-NXT

To remove a container:

docker rm <container_id>

4. Pulling and Listing Docker Images

To pull an image from **Docker Hub**:

docker pull ubuntu

To list downloaded images:

docker images

5. Removing Docker Images

docker rmi <image_id>

CHAPTER 5: CREATING AND MANAGING DOCKER IMAGES

1. Writing a Dockerfile

A **Dockerfile** is a script that automates the **creation of custom Docker images**.

Example: A simple Dockerfile for an **Apache web server**:

Use the base Ubuntu image

FROM ubuntu:latest

Install Apache

RUN apt update && apt install -y apache2

ISDM-NXT

Set the working directory

WORKDIR /var/www/html

Copy website files

COPY index.html /var/www/html/index.html

Expose port 80 for web traffic

EXPOSE 80

Start Apache in foreground

CMD ["apachectl", "-D", "FOREGROUND"]

2. Building a Custom Docker Image

docker build -t my-apache-server .

- -t → Assigns a name to the image (my-apache-server).
- . → Specifies the location of the **Dockerfile**.

3. Running the Custom Image

docker run -d -p 8080:80 my-apache-server

Chapter 6: Docker Networking

1. Viewing Docker Networks

docker network ls

ISDM-NXT

2. Creating a Custom Network

docker network create my-network

3. Connecting Containers to a Network

docker network connect my-network my-container

4. Running Containers in a Custom Network

docker run -d --network=my-network --name container1 nginx

docker run -d --network=my-network --name container2 alpine ping container1

Here, **container2 can communicate with container1** using the container name.

Chapter 7: Using Docker Compose for Multi-Container Applications

1. What is Docker Compose?

Docker Compose **simplifies the management of multi-container applications** by defining services in a docker-compose.yml file.

2. Writing a Docker Compose File

Example: Running a **WordPress site with MySQL**:

```
version: '3'

services:

  db:

    image: mysql:5.7
```

ISDM-NxT

```
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: rootpass
      MYSQL_DATABASE: wordpress
  wordpress:
    image: wordpress:latest
    restart: always
    ports:
      - "8080:80"
    environment:
      WORDPRESS_DB_HOST: db
      WORDPRESS_DB_USER: root
      WORDPRESS_DB_PASSWORD: rootpass
```

3. Running Docker Compose

To start the application:

docker-compose up -d

To stop all services:

docker-compose down

CHAPTER 8: CASE STUDY – DEPLOYING A MICROSERVICES APPLICATION

ISDM-NXT

Scenario:

A **financial services company** wants to deploy a **scalable microservices application** consisting of:

- **User Service** (Python Flask API)
- **Database Service** (MySQL)
- **Frontend Service** (React.js)

Solution Using Docker & Containerization:

1. **Create a Dockerfile** for each microservice.
2. **Define inter-service communication** using Docker networks.
3. **Use Docker Compose to manage all services** in a single YAML file.
4. **Deploy containers on AWS** using Docker Swarm or Kubernetes.

Outcome:

- **Each microservice is deployed in an isolated environment.**
- **Scaling is seamless** since new containers can be launched on demand.
- **CI/CD integration ensures rapid updates** without downtime.

Chapter 9: Exercise

1. **Install Docker and run a sample container.**
2. **Build a Docker image from a Dockerfile.**

ISDM-NxT

3. **Deploy a simple Nginx server using Docker.**
4. **Create a Docker network and connect multiple containers.**
5. **Use Docker Compose to set up a WordPress site.**

CONCLUSION

Docker **revolutionizes application deployment** by enabling **lightweight, scalable, and portable** containerized environments. By mastering **Docker containers, images, networking, and orchestration**, DevOps engineers can **optimize software development workflows and improve deployment efficiency**.

ISDM-NxT

KUBERNETES BASICS

CHAPTER 1: INTRODUCTION TO KUBERNETES

What is Kubernetes?

Kubernetes (K8s) is an **open-source container orchestration platform** that automates the deployment, scaling, and management of containerized applications. It allows developers and DevOps teams to efficiently manage **large-scale, distributed applications** across multiple servers.

Why Use Kubernetes?

- **Automates container deployment** and scaling.
- **Manages multiple containers** across different environments.
- **Ensures high availability** and fault tolerance.
- **Supports rolling updates** and rollback features.
- **Provides built-in service discovery and load balancing.**

Key Features of Kubernetes

- **Self-healing:** Automatically restarts failed containers.
- **Load balancing:** Distributes traffic between multiple containers.
- **Auto-scaling:** Adjusts container instances based on demand.
- **Storage orchestration:** Supports dynamic volume management.
- **Configuration management:** Uses secrets and config maps for environment variables.

ISDM-NxT

Chapter 2: Kubernetes vs. Docker Swarm

Feature	Kubernetes	Docker Swarm
Scalability	Highly scalable	Moderate scalability
Networking	Advanced service discovery	Simple networking
Load Balancing	Built-in load balancer	External load balancer needed
Complexity	Requires more setup	Easier to set up
Use Case	Enterprise-grade orchestration	Small-scale container management

Example Use Case

A **financial company** deploying microservices across multiple cloud servers uses **Kubernetes** to handle scaling, networking, and automated recovery from failures.

Chapter 3: Installing Kubernetes on Linux

1. Installing Kubernetes Tools (kubectl, kubeadm, and kubelet)

On Ubuntu/Debian

sudo apt update

sudo apt install -y kubelet kubeadm kubectl

On CentOS/RHEL

sudo yum install -y kubelet kubeadm kubectl

2. Disabling Swap (Required for Kubernetes)

sudo swapoff -a

sudo sed -i '/ swap / s/^/#/' /etc/fstab

3. Initializing the Kubernetes Cluster (Master Node)

sudo kubeadm init --pod-network-cidr=192.168.1.0/16

After initialization, set up the kubectl configuration:

mkdir -p $HOME/.kube

sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

sudo chown $(id -u):$(id -g) $HOME/.kube/config

4. Installing a Network Plugin (Flannel)

kubectl apply -f
https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml

5. Adding Worker Nodes to the Cluster

On each worker node, run the command generated by kubeadm init:

sudo kubeadm join <master-ip>:6443 --token <token> --discovery-token-ca-cert-hash sha256:<hash>

To verify the worker nodes:

kubectl get nodes

CHAPTER 4: UNDERSTANDING KUBERNETES ARCHITECTURE

1. Kubernetes Components

ISDM-NXT

Component	Description
Master Node	Controls the cluster and manages deployments.
Worker Nodes	Runs application containers.
Pod	Smallest unit in Kubernetes; contains one or more containers.
Deployment	Ensures desired state of application pods.
Service	Exposes an application to other pods or the internet.
Ingress	Manages external access to services.

2. Understanding Pods

Pods are the **smallest deployable unit** in Kubernetes and can contain **one or more containers**.

To create a simple Nginx pod:

kubectl run nginx --image=nginx

To list all pods:

kubectl get pods

Chapter 5: Deploying Applications in Kubernetes

1. Creating a Deployment

Deployments **manage the lifecycle of pods** and ensure availability.

To create an Nginx deployment:

ISDM-NxT

kubectl create deployment nginx-deployment --image=nginx

To scale the deployment:

kubectl scale deployment nginx-deployment --replicas=3

To check deployment status:

kubectl get deployments

2. Exposing a Deployment with a Service

By default, Kubernetes services **assign a stable IP to an application**.

To expose the Nginx deployment on **port 80**:

kubectl expose deployment nginx-deployment --type=NodePort --port=80

To check the service details:

kubectl get svc

CHAPTER 6: KUBERNETES NETWORKING AND LOAD BALANCING

1. Understanding Kubernetes Networking

Kubernetes networking **ensures seamless communication** between pods, nodes, and services.

Networking Type	Purpose
Pod-to-Pod Communication	Allows communication between pods across nodes.
Service-to-Pod Communication	Exposes application pods using services.

ISDM-NXT

Ingress Traffic	Manages external traffic to Kubernetes services.

2. Setting Up an Ingress Controller

Ingress allows **external users** to access Kubernetes applications.

To install an ingress controller:

kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/main/deploy/static/provider/cloud/deploy.yaml

To create an ingress rule:

```
apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

  name: example-ingress

spec:

  rules:

    - host: example.com

      http:

        paths:

          - path: /

            pathType: Prefix

            backend:

              service:
```

ISDM-NXT

```
      name: nginx-deployment
      port:
        number: 80
```

Apply the ingress configuration:

```
kubectl apply -f ingress.yaml
```

CHAPTER 7: KUBERNETES STORAGE AND CONFIGMAPS

1. Using Persistent Volumes

To define a Persistent Volume (PV):

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-pv
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

Apply the configuration:

ISDM-NXT

kubectl apply -f pv.yaml

2. ConfigMaps and Secrets

To create a ConfigMap for environment variables:

kubectl create configmap app-config --from-literal=APP_ENV=production

To use it in a pod:

```
envFrom:
  - configMapRef:
      name: app-config
```

Chapter 8: Case Study – Deploying a Scalable Web Application on Kubernetes

Scenario:

A **tech company** wants to deploy a **scalable e-commerce application** using Kubernetes.

Solution:

1. **Set up a Kubernetes cluster** with kubeadm.
2. **Deploy microservices using Kubernetes deployments** (Frontend, Backend, Database).
3. **Expose services using Kubernetes LoadBalancer and Ingress**.
4. **Use persistent storage for database pods**.
5. **Enable auto-scaling** based on traffic.

ISDM-NXT

Outcome:

- **Application is highly available and can handle traffic spikes.**
- **Pods auto-scale based on demand, ensuring cost efficiency.**
- **Rolling updates and rollbacks minimize downtime.**

Chapter 9: Exercise

1. **Install Kubernetes on a Linux server.**
2. **Create a deployment for an Nginx container.**
3. **Expose the Nginx deployment using a Kubernetes service.**
4. **Configure an Ingress rule to manage external traffic.**
5. **Use a Persistent Volume to store database files.**

Conclusion

Kubernetes **automates the deployment, scaling, and management** of containerized applications. By mastering **pods, deployments, services, networking, and storage**, DevOps engineers can **efficiently run cloud-native applications**.

ISDM-NXT

AUTOMATION WITH ANSIBLE

CHAPTER 1: INTRODUCTION TO ANSIBLE

What is Ansible?

Ansible is an **open-source IT automation tool** that simplifies **server provisioning, configuration management, and application deployment**. It enables **DevOps teams** to automate repetitive tasks and manage infrastructure efficiently.

Why Use Ansible?

- **Agentless:** No need to install software on managed nodes.
- **Simple YAML syntax:** Uses human-readable **Ansible Playbooks**.
- **Scalability:** Manages thousands of servers effortlessly.
- **Idempotent execution:** Ensures tasks run only when needed.
- **Security and flexibility:** Uses **SSH for secure connections**.

Key Features of Ansible

- **Configuration Management:** Automates server setup (e.g., installing Apache).
- **Application Deployment:** Deploys web applications easily.
- **Orchestration:** Manages multi-tier applications across multiple servers.
- **Security Automation:** Ensures compliance and security best practices.

ISDM-NXT

Chapter 2: Installing Ansible on Linux

1. Installing Ansible on Debian/Ubuntu

sudo apt update

sudo apt install -y ansible

2. Installing Ansible on CentOS/RHEL

sudo yum install -y epel-release

sudo yum install -y ansible

3. Verifying Ansible Installation

ansible --version

To check available Ansible modules:

ansible-doc -l

Chapter 3: Ansible Architecture and Components

1. Key Components of Ansible

Component	Description
Control Node	The machine where Ansible is installed.
Managed Nodes	The systems Ansible automates.
Inventory File	A list of target hosts (stored in /etc/ansible/hosts).
Modules	Scripts used to perform automation tasks.

ISDM-NXT

Playbooks	YAML files that define automation tasks.

2. Understanding Ansible Inventory

Ansible uses an **inventory file** (/etc/ansible/hosts) to define target hosts.

Example:

[webservers]

server1 ansible_host=192.168.1.10 ansible_user=root

[dbservers]

server2 ansible_host=192.168.1.20 ansible_user=root

To test connectivity to all servers:

ansible all -m ping

Chapter 4: Running Simple Ansible Commands

1. Running Ad-hoc Commands

Ansible allows running **one-time tasks** using the command-line interface.

Check system uptime:

ansible all -m command -a "uptime"

Install Apache on all web servers:

ansible webservers -m apt -a "name=apache2 state=present" --become

ISDM-NXT

Reboot all managed nodes:

ansible all -m reboot --become

CHAPTER 5: WRITING ANSIBLE PLAYBOOKS

1. What is an Ansible Playbook?

An **Ansible Playbook** is a **YAML file** that defines automation tasks.

Example **Playbook to Install Apache on Web Servers** (install_apache.yml):

```
---

- name: Install Apache Web Server

  hosts: webservers

  become: yes

  tasks:

    - name: Install Apache

      apt:

        name: apache2

        state: present


    - name: Start Apache

      service:

        name: apache2
```

ISDM-NXT

state: started

2. Running an Ansible Playbook

ansible-playbook install_apache.yml

3. Checking Playbook Syntax

Before execution, validate the syntax:

ansible-playbook install_apache.yml --syntax-check

Chapter 6: Configuring Ansible Roles

1. What Are Ansible Roles?

Ansible **roles** allow organizing large Playbooks into reusable components.

To create a new role:

ansible-galaxy init webserver

This creates:

```
webserver/
│ ─── tasks/
│ ─── handlers/
│ ─── templates/
│ ─── vars/
│ ─── defaults/
│ ─── meta/
```

ISDM-NXT

2. Writing an Ansible Role to Install Nginx

Edit the tasks/main.yml file:

```yaml
---
- name: Install Nginx
  apt:
    name: nginx
    state: present

- name: Start Nginx
  service:
    name: nginx
    state: started
```

3. Using the Role in a Playbook

```yaml
---
- name: Deploy Web Server
  hosts: webservers
  become: yes
  roles:
    - webserver
```

4. Running the Role-Based Playbook

```
ansible-playbook deploy_web.yml
```

ISDM-NXT

CHAPTER 7: AUTOMATING SERVER SECURITY WITH ANSIBLE

1. Enforcing Firewall Rules

Playbook to install UFW and configure firewall rules:

```yaml
---
- name: Secure Server with UFW
  hosts: all
  become: yes
  tasks:
    - name: Install UFW
      apt:
        name: ufw
        state: present

    - name: Allow SSH
      ufw:
        rule: allow
        port: 22
        proto: tcp

    - name: Enable UFW
```

ufw:

state: enabled

Run the playbook:

ansible-playbook secure_server.yml

Chapter 8: Case Study – Automating a Web Server Deployment with Ansible

Scenario:

A company needs to **deploy and configure a web server** across multiple Linux instances **without manual intervention**.

Solution Using Ansible:

1. **Define inventory file** with target web servers.
2. **Create an Ansible Playbook** to install and configure Apache.
3. **Use Ansible roles** to organize configurations.
4. **Deploy and validate the server setup using Ansible Playbooks.**

Implementation:

1. Define Inventory (inventory.ini)

[webservers]

server1 ansible_host=192.168.1.10 ansible_user=ubuntu

server2 ansible_host=192.168.1.20 ansible_user=ubuntu

2. Write the Deployment Playbook (deploy_web.yml)

ISDM-NxT

```yaml
---
- name: Deploy Web Server
  hosts: webservers
  become: yes
  tasks:
    - name: Install Apache
      apt:
        name: apache2
        state: present

    - name: Copy Website Files
      copy:
        src: index.html
        dest: /var/www/html/index.html
        mode: '0644'

    - name: Start Apache
      service:
        name: apache2
        state: started
```

3. Run the Playbook to Deploy the Web Server

ISDM-NXT

ansible-playbook -i inventory.ini deploy_web.yml

Outcome:

- **Web servers are configured automatically** without manual intervention.
- **Website files are deployed seamlessly** across all servers.
- **Apache is installed, started, and running on all target machines.**

CHAPTER 9: EXERCISE

1. **Install Ansible on a Linux server.**
2. **Create an inventory file with multiple servers.**
3. **Write a playbook to install MySQL on all database servers.**
4. **Use an Ansible role to manage Nginx installation.**
5. **Automate the setup of user accounts and SSH keys using Ansible.**

CONCLUSION

Ansible simplifies **infrastructure automation** by enabling **consistent, repeatable, and scalable server management**. By mastering **Playbooks, roles, and inventory management**, DevOps engineers can **automate deployments, security configurations, and cloud orchestration** efficiently

ISDM-NXT

Introduction to Cloud (AWS, Google Cloud, Azure)

Chapter 1: Understanding Cloud Computing

What is Cloud Computing?

Cloud computing is the **on-demand delivery of computing resources** such as servers, storage, databases, and networking over the internet. Instead of maintaining physical servers, organizations can leverage cloud providers to scale their infrastructure efficiently.

Key Characteristics of Cloud Computing

- **On-Demand Self-Service:** Users can provision resources instantly.
- **Scalability:** Easily scale infrastructure up or down based on demand.
- **Pay-as-You-Go Pricing:** Users only pay for the resources they consume.
- **Security & Compliance:** Cloud providers offer built-in security measures.
- **Multi-Region Availability:** Cloud services are globally distributed.

Types of Cloud Computing

Type	Description	Example Use Case
Public Cloud	Cloud infrastructure managed by a third-party provider.	Hosting websites, SaaS applications.

ISDM-NxT

Private Cloud	Dedicated cloud resources for a single organization.	Banks, government agencies.
Hybrid Cloud	Combination of public and private cloud environments.	Enterprise data centers with cloud backup.
Multi-Cloud	Use of multiple cloud providers for flexibility.	Disaster recovery and high availability.

Chapter 2: Cloud Service Models

1. Infrastructure as a Service (IaaS)

IaaS provides **virtual machines, storage, and networking** on-demand.

- **Examples:** AWS EC2, Google Compute Engine, Azure Virtual Machines.
- **Use Case:** Running custom applications without managing physical hardware.

2. Platform as a Service (PaaS)

PaaS offers a **fully managed development environment** for building applications.

- **Examples:** AWS Elastic Beanstalk, Google App Engine, Azure App Services.
- **Use Case:** Deploying web applications without managing servers.

3. Software as a Service (SaaS)

ISDM-NXT

SaaS provides **ready-to-use applications** delivered over the internet.

- **Examples:** Google Workspace, Microsoft Office 365, Dropbox.
- **Use Case:** Email, collaboration, and customer relationship management (CRM).

CHAPTER 3: INTRODUCTION TO AMAZON WEB SERVICES (AWS)

1. What is AWS?

AWS (Amazon Web Services) is the **largest cloud provider**, offering over 200 services for computing, storage, networking, and AI.

2. Key AWS Services

Service	Description
EC2 (Elastic Compute Cloud)	Virtual servers in the cloud.
S3 (Simple Storage Service)	Scalable object storage.
RDS (Relational Database Service)	Managed database solutions.
Lambda	Serverless computing for event-driven applications.
VPC (Virtual Private Cloud)	Private cloud networking for AWS resources.

3. Creating an AWS EC2 Instance

1. Log in to **AWS Management Console**.

ISDM-NxT

2. Go to **EC2 → Launch Instance**.
3. Select **Ubuntu** as the operating system.
4. Choose **t2.micro (Free Tier Eligible)**.
5. Configure network settings and **assign a security group**.
6. Click **Launch** and connect via SSH:
7. ssh -i key.pem ubuntu@<EC2_IP>

CHAPTER 4: INTRODUCTION TO GOOGLE CLOUD PLATFORM (GCP)

1. What is Google Cloud?

Google Cloud Platform (GCP) offers **scalable and secure infrastructure** for enterprises. It is known for its **AI/ML, data analytics, and Kubernetes support**.

2. Key Google Cloud Services

Service	Description
Compute Engine	Virtual machines for scalable computing.
Cloud Storage	Secure object storage.
BigQuery	Serverless data warehouse for analytics.
Cloud Functions	Event-driven serverless computing.
Kubernetes Engine	Managed Kubernetes for containerized applications.

3. Creating a Google Cloud VM Instance

1. Sign in to **Google Cloud Console**.

2. Navigate to **Compute Engine → VM Instances**.
3. Click **Create Instance** and select **Machine Type**.
4. Choose **Debian/Ubuntu** as the operating system.
5. Click **Create** and connect using SSH:
6. gcloud compute ssh <instance-name> --zone=<zone>

CHAPTER 5: INTRODUCTION TO MICROSOFT AZURE

1. What is Azure?

Microsoft Azure is a **cloud platform by Microsoft**, offering **hybrid cloud solutions, AI, and DevOps tools**.

2. Key Azure Services

Service	Description
Azure Virtual Machines	Cloud-based virtual servers.
Azure Blob Storage	Scalable object storage.
Azure SQL Database	Fully managed relational databases.
Azure Functions	Serverless computing for automation.
Azure Active Directory	Identity and access management.

3. Creating an Azure Virtual Machine

1. Log in to **Azure Portal**.
2. Go to **Virtual Machines → Create VM**.
3. Select **Ubuntu** and choose **Standard_B1s** size.

ISDM-NXT

4. Configure **Networking and Security Rules**.
5. Click **Create** and connect via SSH:
6. ssh azureuser@<public-ip>

Chapter 6: Cloud Networking and Security

1. Understanding Cloud Networking

- **VPC (AWS), VNet (Azure), and VPC (GCP):** Create isolated network environments.
- **Load Balancers:** Distribute traffic across multiple servers.
- **CDN (Content Delivery Network):** Speeds up content delivery worldwide.

2. Cloud Security Best Practices

- **Use IAM Roles:** Implement least-privilege access.
- **Enable Multi-Factor Authentication (MFA).**
- **Encrypt data using AWS KMS, Google Cloud KMS, or Azure Key Vault.**
- **Monitor cloud activity with CloudTrail, Cloud Logging, or Azure Monitor.**

Chapter 7: Deploying Applications on the Cloud

1. Deploying a Web Application on AWS Using Elastic Beanstalk

eb init -p python-3.7 my-app

ISDM-NXT

```
eb create my-app-env
```

2. Deploying a Web App on Google App Engine

```
gcloud app deploy
```

3. Deploying a Web App on Azure App Services

```
az webapp up --name my-webapp
```

CHAPTER 8: CASE STUDY – MULTI-CLOUD DEPLOYMENT STRATEGY

Scenario:

A global e-commerce company wants a **multi-cloud strategy** to ensure **high availability and disaster recovery**.

Solution Using AWS, Google Cloud, and Azure:

1. **Host primary applications on AWS EC2** for performance.
2. **Store backups on Google Cloud Storage** for redundancy.
3. **Use Azure for analytics and machine learning.**

Outcome:

- **Zero downtime** due to multi-cloud failover.
- **Optimized costs** by choosing the best pricing for each service.
- **Improved security** by leveraging **cloud IAM policies**.

CHAPTER 9: EXERCISE

1. **Create an AWS EC2 instance and configure a web server.**

ISDM-NxT

2. **Deploy a virtual machine on Google Cloud Compute Engine.**
3. **Set up an Azure Virtual Machine with Linux OS.**
4. **Implement IAM policies to restrict user access.**
5. **Use AWS S3, Google Cloud Storage, or Azure Blob Storage to store files.**

CONCLUSION

Cloud computing provides **scalable, cost-effective, and secure solutions** for modern applications. By mastering **AWS, Google Cloud, and Azure**, DevOps engineers can **deploy, manage, and optimize cloud-based applications efficiently**.

ISDM-NXT

LINUX IN CLOUD ENVIRONMENTS

CHAPTER 1: INTRODUCTION TO LINUX IN CLOUD COMPUTING

What is Cloud Computing?

Cloud computing is the **on-demand delivery of computing resources** such as servers, storage, databases, networking, and applications over the internet. It eliminates the need for maintaining physical data centers and provides **scalability, flexibility, and cost-efficiency**.

Why is Linux the Preferred OS in Cloud Environments?

Linux is the **dominant operating system in cloud computing** because of its:

- **Open-source nature** – Free and highly customizable.
- **Stability and security** – Essential for enterprise workloads.
- **Lightweight design** – Optimized for performance and efficiency.
- **Compatibility** – Supports major cloud providers like **AWS, Google Cloud, and Azure**.

Role of Linux in Cloud Environments

- **Runs cloud-based virtual machines and containers.**
- **Powers Kubernetes clusters for container orchestration.**
- **Supports automation and DevOps workflows (e.g., Terraform, Ansible).**
- **Enables scalable web applications and database services.**

ISDM-NxT

CHAPTER 2: LINUX DISTRIBUTIONS USED IN CLOUD ENVIRONMENTS

Distribution	Cloud Provider Support	Use Case
Ubuntu Server	AWS, Azure, GCP	General-purpose cloud servers, DevOps workloads
Amazon Linux	AWS only	Optimized for AWS EC2 instances
Red Hat Enterprise Linux (RHEL)	AWS, Azure, GCP	Enterprise applications, security compliance
Debian	AWS, GCP, Azure	Lightweight and secure cloud environments
Alpine Linux	Docker, Kubernetes	Minimalist Linux for containers

Example Use Case

A **startup deploying a web application on AWS** chooses **Ubuntu Server** due to its **ease of use and large community support**.

CHAPTER 3: SETTING UP LINUX VIRTUAL MACHINES IN THE CLOUD

1. Creating an AWS EC2 Instance with Linux

1. Log in to **AWS Management Console**.
2. Navigate to **EC2 Dashboard** → Click **Launch Instance**.
3. Select **Ubuntu 22.04 LTS** as the OS.

ISDM-NxT

4. Choose **t2.micro (Free Tier Eligible)** instance type.
5. Configure networking and **assign a security group**.
6. Click **Launch**, then connect via SSH:
7. ssh -i key.pem ubuntu@<EC2_Public_IP>

2. Creating a Google Cloud Compute Engine VM

1. Open **Google Cloud Console** → Go to **Compute Engine**.
2. Click **Create Instance**.
3. Select **Debian or Ubuntu** as the OS.
4. Choose **n1-standard-1** machine type.
5. Click **Create** and connect using SSH:
6. gcloud compute ssh instance-name --zone=us-central1-a

3. Creating an Azure Virtual Machine

1. Log in to **Azure Portal** → Go to **Virtual Machines**.
2. Click **Create VM** → Select **Ubuntu** as the OS.
3. Configure instance size and **set up security rules**.
4. Click **Create** and connect via SSH:
5. ssh azureuser@<Public-IP>

CHAPTER 4: LINUX SERVER ADMINISTRATION IN THE CLOUD

1. Managing Users and Permissions

To create a new user:

sudo adduser devops

To grant **sudo privileges**:

sudo usermod -aG sudo devops

2. Setting Up SSH Key Authentication

To generate an SSH key pair:

ssh-keygen -t rsa -b 4096

To copy the key to a cloud server:

ssh-copy-id user@<server-ip>

3. Configuring a Firewall (UFW on Ubuntu/Debian)

To allow SSH, HTTP, and HTTPS traffic:

sudo ufw allow OpenSSH

sudo ufw allow 80/tcp

sudo ufw allow 443/tcp

sudo ufw enable

4. Automating Updates and Security Patches

To enable automatic updates:

sudo apt install unattended-upgrades -y

sudo dpkg-reconfigure unattended-upgrades

Chapter 5: Linux Storage and File Systems in Cloud

1. Cloud Storage Options

ISDM-NXT

Storage Type	Cloud Provider Example	Use Case
Block Storage	AWS EBS, Google Persistent Disk, Azure Disk Storage	Virtual machine disks
Object Storage	AWS S3, Google Cloud Storage, Azure Blob Storage	Storing files, backups, images
File Storage	AWS EFS, Google Filestore, Azure Files	Shared file systems

2. Mounting a Cloud Storage Disk on Linux

AWS EBS Volume (Example for Ubuntu EC2 Instance)

1. Attach a new EBS volume to an EC2 instance.
2. Format and mount the volume:
3. sudo mkfs.ext4 /dev/xvdf
4. sudo mkdir /mnt/data
5. sudo mount /dev/xvdf /mnt/data
6. To make it persistent:
7. echo "/dev/xvdf /mnt/data ext4 defaults,nofail 0 2" | sudo tee -a /etc/fstab

---

CHAPTER 6: LINUX IN CLOUD AUTOMATION & DEVOPS

1. Automating Cloud Infrastructure with Terraform

Terraform allows infrastructure to be managed as **code (IaC)**.

Example: Provision an AWS EC2 instance using Terraform:

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


resource "aws_instance" "web" {
  ami           = "ami-0abcdef1234567890"
  instance_type = "t2.micro"
}
```

To deploy:

```
terraform init
terraform apply -auto-approve
```

2. Automating Configuration with Ansible

Example: Install Apache on all cloud servers using Ansible:

```
---
- name: Install Apache Web Server
  hosts: cloud_servers
  become: yes
  tasks:
    - name: Install Apache
      apt:
```

ISDM-NXT

name: apache2

state: present

Run the playbook:

ansible-playbook install_apache.yml

CHAPTER 7: MONITORING AND SECURITY IN LINUX CLOUD ENVIRONMENTS

1. Cloud Monitoring Tools

Cloud Provider	Monitoring Tool	Purpose
AWS	CloudWatch	Logs, metrics, and alerts
Google Cloud	Operations Suite	Resource monitoring
Azure	Azure Monitor	Performance and security logs

2. Using Log Files to Monitor Linux Instances

To check system logs on a Linux cloud instance:

sudo journalctl -xe

sudo tail -f /var/log/syslog

3. Implementing Cloud Security Best Practices

- **Enable Multi-Factor Authentication (MFA)** for SSH access.
- **Use IAM roles instead of hardcoding credentials.**
- **Encrypt sensitive data at rest and in transit.**
- **Use security groups and firewalls to limit access.**

ISDM-NxT

Chapter 8: Case Study – Deploying a Web Application on AWS Using Linux

Scenario:

A **SaaS company** wants to deploy a **highly available web application** on AWS using Linux instances.

Solution Using AWS and Linux:

1. **Launch AWS EC2 instances running Ubuntu.**
2. **Install and configure Nginx as a web server.**
3. **Use AWS Elastic Load Balancer to distribute traffic.**
4. **Store static assets in AWS S3.**
5. **Monitor logs using AWS CloudWatch.**

Outcome:

- **Web application is scalable and fault-tolerant.**
- **Nginx runs efficiently on Linux instances.**
- **Cloud security policies protect against cyber threats.**

ISDM-NXT

Chapter 9: Exercise

1. **Deploy an EC2 instance and configure Apache.**
2. **Mount a cloud storage volume to a Linux VM.**
3. **Automate server updates using Ansible.**

4. **Set up a firewall to allow only SSH and HTTP traffic.**
5. **Monitor server logs and create an alerting system.**

CONCLUSION

Linux is the **backbone of cloud environments**, enabling **scalable, automated, and secure computing**. Mastering **Linux for cloud computing** is essential for **DevOps engineers, system administrators, and cloud architects**.

ISDM-NXT

CI/CD PIPELINE WITH LINUX

CHAPTER 1: INTRODUCTION TO CI/CD PIPELINES

What is CI/CD?

CI/CD (Continuous Integration and Continuous Deployment/Delivery) is a **DevOps practice** that automates the process of **building, testing, and deploying software**. It ensures faster and more reliable releases by **reducing manual intervention** and improving efficiency.

Why Use CI/CD in Software Development?

- **Automates code testing and integration.**
- **Reduces deployment errors and manual work.**
- **Ensures faster and reliable software releases.**
- **Improves collaboration between developers and operations teams.**

CI/CD Pipeline Components

Component	Description
Continuous Integration (CI)	Automates code merging, builds, and testing.
Continuous Deployment (CD)	Deploys changes automatically to production.
Continuous Delivery	Prepares code for deployment but requires manual approval.
Monitoring & Feedback	Tracks application performance after deployment.

ISDM-NXT

CHAPTER 2: SETTING UP A CI/CD PIPELINE ON LINUX

1. Installing Required Tools

For setting up a CI/CD pipeline on **Linux**, we need:

- **Git** – Version control system.
- **Jenkins** – CI/CD automation tool.
- **Docker** – Containerization for deployment.
- **Ansible** – Configuration management.

Installing Git on Linux

sudo apt install git -y # Debian/Ubuntu

sudo yum install git -y # CentOS/RHEL

Installing Jenkins on Linux

sudo apt update

sudo apt install openjdk-11-jdk -y

wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key | sudo apt-key add -

sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'

sudo apt update

sudo apt install jenkins -y

sudo systemctl enable --now jenkins

Installing Docker on Linux

ISDM-NXT

sudo apt install docker.io -y

sudo systemctl enable --now docker

Installing Ansible on Linux

sudo apt install ansible -y

Chapter 3: Creating a Git Repository for CI/CD

1. Initializing a Git Repository

mkdir ci-cd-project

cd ci-cd-project

git init

2. Creating a Simple Python Application

Create a Python script app.py:

print("Hello, CI/CD Pipeline with Linux!")

3. Creating a GitHub Repository

1. Create a repository on **GitHub**.
2. Add the remote repository:
3. git remote add origin https://github.com/user/ci-cd-project.git
4. git add .
5. git commit -m "Initial commit"
6. git push origin main

ISDM-NXT

CHAPTER 4: CONFIGURING JENKINS FOR CI/CD

1. Setting Up Jenkins Pipeline

1. Open **Jenkins Web UI** (http://<server-ip>:8080).
2. Go to **Manage Jenkins** → **Plugins** → Install **Pipeline Plugin**.
3. Create a **New Item** → Choose **Pipeline**.
4. In the **Pipeline Script**, enter:

```
pipeline {
  agent any
  stages {
    stage('Clone Repository') {
      steps {
        git 'https://github.com/user/ci-cd-project.git'
      }
    }
    stage('Build') {
      steps {
        sh 'echo "Building application..."'
      }
    }
    stage('Test') {
      steps {
```

ISDM-NXT

```
        sh 'python3 -m unittest discover tests'
      }
    }
    stage('Deploy') {
      steps {
        sh 'ansible-playbook deploy.yml'
      }
    }
  }
}
```

5. Save and run the pipeline.

Chapter 5: Automating Deployment with Ansible

1. Writing an Ansible Playbook for Deployment

Create a file deploy.yml:

```
---
- name: Deploy Application
  hosts: webserver
  become: yes
  tasks:
    - name: Copy application files
```

ISDM-NXT

```
    copy:
      src: /var/lib/jenkins/workspace/ci-cd-project
      dest: /var/www/html/
  - name: Restart Apache
    service:
      name: apache2
      state: restarted
```

2. Running the Playbook

```
ansible-playbook -i inventory deploy.yml
```

CHAPTER 6: USING DOCKER FOR CONTAINERIZED DEPLOYMENT

1. Creating a Dockerfile

```
FROM python:3.9
WORKDIR /app
COPY . .
CMD ["python3", "app.py"]
```

2. Building and Running a Docker Container

```
docker build -t myapp .
docker run -d --name myapp -p 5000:5000 myapp
```

3. Integrating Docker into Jenkins Pipeline

Modify Jenkinsfile:

ISDM-NXT

```
stage('Docker Build and Run') {
   steps {
      sh 'docker build -t myapp .'
      sh 'docker run -d --name myapp -p 5000:5000 myapp'
   }
}
```

CHAPTER 7: IMPLEMENTING CONTINUOUS MONITORING IN CI/CD

1. Monitoring Jenkins Logs

```
tail -f /var/log/jenkins/jenkins.log
```

2. Monitoring Application Logs with Docker

```
docker logs -f myapp
```

3. Using Prometheus and Grafana for Monitoring

To install Prometheus:

```
sudo apt install prometheus -y
```

To install Grafana:

```
sudo apt install grafana -y
```

CHAPTER 8: CASE STUDY – IMPLEMENTING CI/CD FOR A WEB APPLICATION

Scenario:

A **DevOps team** wants to automate the deployment of a **Flask web application** using **Jenkins, Docker, and Ansible**.

Solution Using CI/CD Pipeline:

1. **Developers push code to GitHub.**
2. **Jenkins pulls the latest code and runs tests.**
3. **Docker builds and deploys a containerized application.**
4. **Ansible configures the web server and deploys the application.**
5. **Prometheus monitors application performance.**

Outcome:

- **Automated deployments ensure reliability.**
- **Reduced deployment time from hours to minutes.**
- **Scalability and flexibility improved through containerization.**

CHAPTER 9: EXERCISE

1. **Install Jenkins and set up a CI/CD pipeline.**
2. **Create a GitHub repository and connect it to Jenkins.**
3. **Write an Ansible playbook to deploy a web application.**
4. **Containerize an application using Docker and deploy it via Jenkins.**
5. **Monitor the CI/CD pipeline logs and optimize performance.**

ISDM-NxT

CONCLUSION

CI/CD pipelines enable **efficient, automated software delivery** using **Linux-based tools like Jenkins, Docker, and Ansible**. Mastering **CI/CD in Linux environments** is essential for **DevOps engineers** to **streamline software development and deployment**.

ISDM-NXT

INFRASTRUCTURE AS CODE (TERRAFORM BASICS)

CHAPTER 1: INTRODUCTION TO INFRASTRUCTURE AS CODE (IAC)

What is Infrastructure as Code (IaC)?

Infrastructure as Code (**IaC**) is a **DevOps practice** that automates the provisioning, management, and deployment of infrastructure using **declarative code**. Instead of manually setting up cloud resources, IaC allows teams to **define infrastructure in code**, making it **repeatable, consistent, and scalable**.

Why Use IaC?

- **Automates infrastructure provisioning** and reduces manual errors.
- **Ensures consistency across environments** (development, staging, production).
- **Enables version control for infrastructure** using Git.
- **Enhances scalability by automating resource allocation** in the cloud.

Popular IaC Tools

Tool	Description	Use Case
Terraform	Open-source tool for provisioning infrastructure across multiple cloud providers.	AWS, GCP, Azure automation

ISDM-NxT

AWS CloudFormation	AWS-native IaC tool for managing AWS resources.	AWS infrastructure management
Ansible	Used for configuration management and infrastructure provisioning.	Application deployment
Pulumi	Uses programming languages for defining infrastructure.	IaC with Python, TypeScript

---

CHAPTER 2: WHAT IS TERRAFORM?

1. Understanding Terraform

Terraform is an **open-source Infrastructure as Code (IaC) tool** that allows developers to **define and manage cloud resources** using **a declarative configuration language (HCL – HashiCorp Configuration Language)**.

2. Why Use Terraform?

- **Cloud-agnostic:** Works with AWS, Google Cloud, Azure, and more.
- **Declarative syntax:** Define infrastructure in a simple, readable format.
- **Automated provisioning:** Automatically creates, updates, and destroys resources.
- **State management:** Keeps track of infrastructure changes.
- **Scalability:** Easily scales up or down as needed.

ISDM-NxT

3. Terraform vs. Other IaC Tools

Feature	Terraform	CloudFormation	Ansible
Multi-cloud support	✅ Yes	❌ No (AWS only)	✅ Yes
Declarative syntax	✅ Yes	✅ Yes	❌ No (Procedural)
State Management	✅ Yes	✅ Yes	❌ No
Provisioning & Configuration	✅ Yes	✅ Yes	✅ Yes

CHAPTER 3: INSTALLING TERRAFORM ON LINUX

1. Download and Install Terraform

On Ubuntu/Debian:

sudo apt update

sudo apt install -y wget unzip

wget https://releases.hashicorp.com/terraform/1.5.5/terraform_1.5.5_linux_amd64.zip

unzip terraform_1.5.5_linux_amd64.zip

sudo mv terraform /usr/local/bin/

terraform --version

On CentOS/RHEL:

ISDM-NxT

sudo yum install -y wget unzip

wget
https://releases.hashicorp.com/terraform/1.5.5/terraform_1.5.5_linux_amd64.zip

unzip terraform_1.5.5_linux_amd64.zip

sudo mv terraform /usr/local/bin/

terraform --version

CHAPTER 4: WRITING YOUR FIRST TERRAFORM CONFIGURATION

1. Setting Up a Terraform Project

Create a new directory and navigate into it:

mkdir terraform-project

cd terraform-project

2. Writing a Terraform Configuration File

Create a file called main.tf:

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


resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
```

ISDM-NXT

```
 tags = {
   Name = "TerraformInstance"
 }
}
```

This configuration:

- Defines AWS as the **cloud provider**.
- Creates an **EC2 instance** with a specific Amazon Machine Image (AMI).
- Assigns a **name tag** to the instance.

CHAPTER 5: INITIALIZING AND APPLYING TERRAFORM CONFIGURATION

1. Initializing Terraform

Before applying changes, initialize Terraform:

```
terraform init
```

This command:

- Downloads necessary **Terraform providers**.
- Sets up the Terraform working directory.

2. Creating an Execution Plan

Check what Terraform will create:

```
terraform plan
```


ISDM-NXT

3. Applying the Configuration

Deploy the infrastructure:

terraform apply -auto-approve

This command provisions an **EC2 instance on AWS**.

4. Verifying the Deployed Instance

To check if the instance is running:

aws ec2 describe-instances --query 'Reservations[*].Instances[*].InstanceId'

CHAPTER 6: MANAGING TERRAFORM STATE AND DESTROYING RESOURCES

1. Understanding Terraform State

Terraform maintains a **state file (terraform.tfstate)** that keeps track of deployed infrastructure.

To inspect the state file:

terraform show

2. Destroying Infrastructure

To delete all resources managed by Terraform:

terraform destroy -auto-approve

CHAPTER 7: USING VARIABLES AND OUTPUTS IN TERRAFORM

1. Defining Variables

ISDM-NXT

Variables make Terraform configurations **dynamic and reusable**.

Define a variables file (variables.tf):

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

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

Modify main.tf to use these variables:

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

resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = var.instance_type

  tags = {
    Name = "TerraformInstance"
  }
```

ISDM-NXT

}

To apply changes with variables:

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

2. Using Outputs to Display Information

Define an **output** in outputs.tf:

output "instance_public_ip" {

  value = aws_instance.web.public_ip

}

After applying the configuration, Terraform will display the **public IP** of the created instance.

CHAPTER 8: CASE STUDY – DEPLOYING A WEB SERVER USING TERRAFORM

Scenario:

A **startup wants to automate the deployment of a web server** on AWS using Terraform.

Solution Using Terraform:

1. **Define an EC2 instance** with Terraform.
2. **Use a user data script** to install and start Apache automatically.
3. **Create a security group** to allow HTTP traffic.

Implementation

ISDM-NXT

Modify main.tf:

```
resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  user_data = <<-EOF
              #!/bin/bash
              sudo apt update
              sudo apt install -y apache2
              sudo systemctl start apache2
              EOF

  tags = {
    Name = "TerraformWebServer"
  }
}
```

Outcome:

- **Fully automated web server deployment.**
- **Infrastructure as Code for repeatability.**
- **Easier scaling and management of cloud resources.**

ISDM-NXT

CHAPTER 9: EXERCISE

1. **Install Terraform and initialize a new project.**
2. **Create a Terraform configuration to deploy an AWS EC2 instance.**
3. **Modify the Terraform configuration to use variables.**
4. **Add an output variable to display the public IP of the instance.**
5. **Destroy the infrastructure and clean up resources.**

CONCLUSION

Terraform provides **powerful, repeatable, and scalable infrastructure automation** for cloud environments. By mastering **Terraform basics**, DevOps engineers can **provision, manage, and deploy cloud infrastructure efficiently**.

ISDM-NXT

Assignment Solution: Deploy a Simple Docker Container on a Linux System – Step-by-Step Guide

Objective

This assignment provides a **step-by-step guide** to deploying a simple Docker container on a **Linux system**. The goal is to **install Docker, run a basic container**, and understand fundamental Docker commands.

Step 1: Install Docker on Linux

1. Update the Package Repository

Before installing Docker, update the system package repository:

sudo apt update -y # Ubuntu/Debian

sudo yum update -y # CentOS/RHEL

2. Install Docker

On Ubuntu/Debian

sudo apt install -y docker.io

On CentOS/RHEL

sudo yum install -y docker

3. Enable and Start the Docker Service

Once installed, start and enable the Docker service:

sudo systemctl start docker

sudo systemctl enable docker

4. Verify Docker Installation

To ensure Docker is running correctly:

docker --version

sudo systemctl status docker

STEP 2: RUN A SIMPLE DOCKER CONTAINER

1. Pull a Docker Image

Docker uses **pre-built images** to run containers. We will use the **hello-world** image to verify the installation:

docker pull hello-world

2. Run the Docker Container

docker run hello-world

- This command **downloads the image (if not already present)** and **runs it as a container**.
- It should display a message:
- Hello from Docker!
- This message shows that your installation appears to be working correctly.

STEP 3: RUNNING A WEB SERVER CONTAINER (NGINX)

1. Pull the Nginx Image

ISDM-NXT

Nginx is a **lightweight web server** commonly used for serving static websites and reverse proxying:

docker pull nginx

2. Run the Nginx Container

docker run -d -p 8080:80 --name mynginx nginx

Explanation:

- -d → Runs the container in **detached mode** (background).
- -p 8080:80 → Maps **port 8080 on the host to port 80** inside the container.
- --name mynginx → Assigns a **name to the container**.

3. Verify the Running Container

To check if the container is running:

docker ps

4. Access the Web Server

Open a browser and navigate to:

http://<your-linux-ip>:8080

or use:

curl http://localhost:8080

You should see the **default Nginx welcome page**.

STEP 4: MANAGING DOCKER CONTAINERS

1. Listing Running Containers

ISDM-NXT

docker ps

To list **all containers (including stopped ones)**:

docker ps -a

2. Stopping a Running Container

To stop the Nginx container:

docker stop mynginx

3. Restarting a Container

docker start mynginx

4. Removing a Container

To delete a stopped container:

docker rm mynginx

5. Removing an Image

To remove the Nginx image:

docker rmi nginx

ISDM-NXT

STEP 5: DEPLOYING A CUSTOM DOCKER CONTAINER WITH A WEBSITE

1. Create a Custom HTML File

mkdir website

cd website

echo "<h1>Welcome to My Docker Web Server</h1>" > index.html

2. Write a Dockerfile

Create a file named Dockerfile in the website directory:

```
FROM nginx:latest

COPY index.html /usr/share/nginx/html/index.html
```

3. Build a Custom Docker Image

```
docker build -t mynginx-image .
```

- -t mynginx-image → Tags the image as mynginx-image.
- . → Refers to the current directory containing the Dockerfile.

4. Run the Custom Container

```
docker run -d -p 8080:80 --name custom-nginx mynginx-image
```

5. Access the Custom Website

Visit:

http://localhost:8080

You should see:

Welcome to My Docker Web Server

Step 6: Making Docker Containers Persistent

1. Mounting a Volume for Data Persistence

Docker volumes ensure data **persists** even if a container is deleted.

```
docker run -d -p 8080:80 -v
/home/user/website:/usr/share/nginx/html --name persistent-nginx
nginx
```

ISDM-NXT

Now, any changes made in /home/user/website/ will reflect in the container.

Case Study – Deploying a Simple Web App in a Docker Container

Scenario:

A **startup** needs a **lightweight web server** running inside a **Docker container** for hosting a company webpage.

Solution Using Docker:

1. Install Docker on a **Linux server**.
2. Pull the **Nginx image** and run it as a **container**.
3. Create a **custom HTML page** and deploy it using a **custom Docker image**.
4. Ensure **data persistence** using **Docker volumes**.

Outcome:

- **Web application runs inside a container**, reducing **setup time**.
- **Changes to website files can be reflected instantly** using mounted volumes.
- **Infrastructure is portable**, allowing easy deployment on **any cloud platform**.

STEP 7: EXERCISE

1. **Install Docker on a Linux system.**

ISDM-NXT

2. **Run a simple "hello-world" container.**
3. **Deploy an Nginx container and access the web server.**
4. **Create a custom Dockerfile and build a new image.**
5. **Mount a volume to persist website files inside a Docker container.**
6. **Remove all stopped containers and unused images.**

CONCLUSION

By following this guide, you have successfully:

✅ **Installed and configured Docker on a Linux system.**
✅ **Deployed a basic "hello-world" container.**
✅ **Launched an Nginx web server using Docker.**
✅ **Created a custom Docker image for a simple website.**
✅ **Managed and removed Docker containers and images.**

ISDM-NxT

ASSIGNMENT SOLUTION: AUTOMATE SERVER CONFIGURATION USING ANSIBLE – STEP-BY-STEP GUIDE

Objective

This assignment provides a **step-by-step guide** to automating server configuration using **Ansible**. The goal is to **install Ansible, configure an inventory file, write an Ansible Playbook**, and execute automation tasks such as **installing software, managing users, and configuring services**.

STEP 1: INSTALL ANSIBLE ON A LINUX CONTROL NODE

1. Update the System Packages

Before installing Ansible, update your package manager:

sudo apt update -y   # Ubuntu/Debian

sudo yum update -y   # CentOS/RHEL

2. Install Ansible

On Ubuntu/Debian:

sudo apt install ansible -y

On CentOS/RHEL:

sudo yum install ansible -y

3. Verify Ansible Installation

ansible --version

ISDM-NxT

Step 2: Configure Ansible Inventory File

Ansible requires an **inventory file** to define managed hosts.

1. Locate the Default Inventory File

cat /etc/ansible/hosts

2. Create a Custom Inventory File

Create a new inventory file:

sudo nano /etc/ansible/inventory

Add the following content:

[webservers]

server1 ansible_host=192.168.1.10 ansible_user=ubuntu ansible_ssh_private_key_file=~/.ssh/id_rsa

[dbservers]

server2 ansible_host=192.168.1.20 ansible_user=ubuntu ansible_ssh_private_key_file=~/.ssh/id_rsa

3. Test Ansible Connectivity

To check if Ansible can communicate with the servers:

ansible all -m ping -i /etc/ansible/inventory

If successful, it should return:

server1 | SUCCESS => { "ping": "pong" }

server2 | SUCCESS => { "ping": "pong" }

ISDM-NXT

Step 3: Write an Ansible Playbook for Server Configuration

An **Ansible Playbook** is a **YAML file** that defines automation tasks.

1. Create an Ansible Playbook

Create a file named server_config.yml:

```
nano server_config.yml
```

Add the following **YAML configuration**:

```
---
- name: Automate Server Configuration
  hosts: all
  become: yes
  tasks:

    - name: Update package repositories
      apt:
        update_cache: yes
      when: ansible_os_family == "Debian"

    - name: Install required packages
      apt:
        name:
```

ISDM-NXT

```
    - vim
    - git
    - curl
   state: present
  when: ansible_os_family == "Debian"

- name: Create a new user
  user:
   name: devops
   password: "{{ 'devops_password' | password_hash('sha512') }}"
   shell: /bin/bash
   state: present

- name: Add user to sudo group
  user:
   name: devops
   groups: sudo
   append: yes

- name: Configure SSH for secure access
  lineinfile:
```

ISDM-NXT

```
      path: /etc/ssh/sshd_config
      regexp: "^PermitRootLogin"
      line: "PermitRootLogin no"
    notify:
      - Restart SSH

handlers:
  - name: Restart SSH
    service:
      name: ssh
      state: restarted
```

2. Explanation of Playbook Tasks

- **Updates system package repositories** (if on Debian-based systems).
- **Installs essential packages** (vim, git, curl).
- **Creates a new user (devops) with a secure password**.
- **Adds the user to the sudo group** for administrative privileges.
- **Modifies SSH configuration** to **disable root login** for security.
- **Uses a handler** to **restart SSH** if configuration changes.

STEP 4: RUN THE ANSIBLE PLAYBOOK

ISDM-NXT

1. Execute the Playbook

To apply server configurations, run:

ansible-playbook -i /etc/ansible/inventory server_config.yml

2. Expected Output

You should see output similar to:

PLAY [Automate Server Configuration]
****

TASK [Update package repositories]
****

ok: [server1]

ok: [server2]

TASK [Install required packages]
****

changed: [server1]

changed: [server2]

TASK [Create a new user]
****
**

changed: [server1]

changed: [server2]

TASK [Configure SSH for secure access]
****

changed: [server1]

changed: [server2]

TASK [Restart SSH]
****

changed: [server1]

changed: [server2]

PLAY RECAP
****

server1 : ok=5 changed=4

server2 : ok=5 changed=4

Step 5: Verify Configuration Changes

1. Check Installed Packages

dpkg -l | grep vim # Ubuntu/Debian

2. Verify the New User

cat /etc/passwd | grep devops

3. Confirm SSH Configuration

cat /etc/ssh/sshd_config | grep PermitRootLogin

Expected Output:

PermitRootLogin no

STEP 6: AUTOMATING MULTIPLE SERVER CONFIGURATIONS

Modify server_config.yml to include:

- **Web Server Setup (Apache/Nginx)**
- **Database Configuration (MySQL/PostgreSQL)**
- **Firewall Configuration (UFW/IPTables)**

Example: Install and start Apache web server:

```
- name: Install Apache Web Server
  apt:
    name: apache2
    state: present
  when: ansible_os_family == "Debian"

- name: Start Apache Service
  service:
    name: apache2
    state: started
```

ISDM-NXT

enabled: yes

Run the updated playbook:

ansible-playbook -i /etc/ansible/inventory server_config.yml

Case Study – Automating Server Configuration for a Web Application

Scenario:

A **company needs to set up multiple Linux servers** with:

1. **Security settings** (Disable root login, add sudo user).
2. **Software installations** (Web server, Git, Curl).
3. **Automated firewall rules**.

Solution Using Ansible:

1. **Define the inventory file** with target servers.
2. **Create a Playbook** to configure users, SSH, install software.
3. **Execute the Playbook across multiple servers** for consistency.

Outcome:

- **All servers are configured identically, reducing manual effort**.
- **Security best practices are enforced automatically**.
- **Deployment time is reduced from hours to minutes**.

ISDM-NxT

STEP 7: EXERCISE

1. **Install Ansible and set up an inventory file.**
2. **Write a Playbook to create a new user and install necessary packages.**
3. **Modify the SSH configuration to disable root login.**
4. **Deploy a web server using Ansible.**
5. **Verify that all changes have been applied correctly.**

CONCLUSION

By following this guide, you have successfully:

✅ **Installed and configured Ansible** on a Linux system.

✅ **Defined an inventory file** to manage multiple servers.

✅ **Written an Ansible Playbook** to automate server configurations.

✅ **Executed the Playbook to apply system changes.**

ISDM-NXT

ISDM-NXT