1. Introduction of Virtualization

1.1 Definition and Basic Concept

Virtualization is a technology that enables the creation of **multiple simulated (virtual) computing environments** from a **single physical hardware system**. It abstracts physical resources (CPU, memory, storage, network) and presents them as logical resources.

In virtualization:

- Physical hardware → abstracted by software
- Multiple Virtual Machines (VMs) → run independently
- Each VM behaves like a real computer

1.2 Purpose and Need

- Efficient utilization of hardware
- Reduction in hardware costs
- Server consolidation
- Isolation between applications
- Faster deployment and scalability
- Disaster recovery and backup
- Testing and development environments

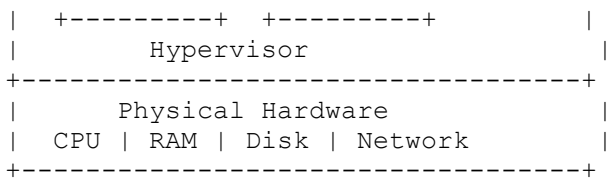
1.3 Architecture and Components

Core Components

1. **Physical Host**
2. **Hypervisor**
3. **Virtual Machines**
4. **Guest Operating Systems**
5. **Virtual Hardware (vCPU, vRAM, vDisk)**

Text-Based Architecture Diagram

```
+-----------------------------------+
|        Virtual Machines           |
|  +---------+  +---------+         |
|  | Guest OS |  | Guest OS |         |
```



1.4 Working / Workflow

1. Hypervisor loads on hardware
2. Hypervisor abstracts physical resources
3. Virtual machines request resources
4. Hypervisor schedules and allocates resources
5. VMs operate independently

1.5 Types and Classifications

- Server Virtualization
- Desktop Virtualization
- Storage Virtualization
- Network Virtualization
- Application Virtualization
- OS-level Virtualization

1.6 Features

- Hardware abstraction
- Isolation
- Resource pooling
- Portability
- Scalability
- High availability support

1.7 Advantages

- Cost reduction
- Better resource utilization
- Easy backup and recovery

- Faster provisioning
- Improved system reliability

1.8 Disadvantages

- Performance overhead
- Licensing costs
- Complexity in management
- Hardware dependency

1.9 Real-World Use Cases

- Cloud computing (AWS, Azure)
- Data centers
- Software testing labs
- Disaster recovery sites

1.10 Important Exam Points and Keywords

Keywords: Hypervisor, VM, Host, Guest OS, Abstraction, Resource Pooling

---

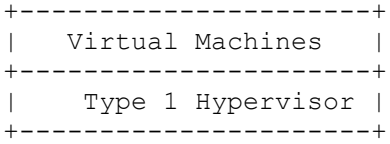
2. Virtualization Types

2.1 Type 1 – Bare Metal Hypervisor

Definition

A hypervisor that runs **directly on physical hardware** without a host OS.

Architecture



```
|   Physical Hardware  |
+----------------------+
```

Features

- High performance
- Direct hardware access
- Enterprise-grade

Examples

- VMware ESXi
- Microsoft Hyper-V Server
- Xen

2.2 Type 2 – Hosted Hypervisor

Definition

A hypervisor that runs **on top of a host operating system**.

Architecture

```
+----------------------+
|   Virtual Machines   |
+----------------------+
|   Type 2 Hypervisor  |
+----------------------+
|   Host Operating OS  |
+----------------------+
|   Physical Hardware  |
+----------------------+
```

Features

- Easy installation
- Suitable for desktops
- Less performance

Examples

- VirtualBox
- VMware Workstation
- Parallels Desktop

Comparison Table

Feature	Type 1	Type 2
Performance	High	Medium
Host OS	No	Yes
Use Case	Data Centers	Desktop
Security	High	Lower

---

3. Virtualization Techniques

3.1 Full Virtualization

Definition

Guest OS runs **unmodified** as if on real hardware.

Working

- Hypervisor emulates complete hardware
- Traps sensitive instructions

Advantages

- No OS modification
- Broad OS support

Disadvantages

- Performance overhead

3.2 Hardware Virtualization

Definition

Uses **CPU virtualization extensions** (Intel VT-x, AMD-V).

Architecture

```
CPU with VT-x / AMD-V
↓
Hypervisor
↓
Virtual Machines
```

Features

- Near-native performance
- Reduced overhead

3.3 Para-Virtualization

Definition

Guest OS is **modified** to interact with hypervisor.

Working

- Guest OS calls hypervisor directly
- Uses hypercalls

Advantages

- Better performance
- Lower overhead

Disadvantages

- Requires OS modification

Comparison Table

Technique	OS Modified	Performance
Full	No	Medium
Hardware	No	High
Para	Yes	Very High

---

4. Virtual Machine Management

4.1 Cloning

Definition

Creating an **exact copy** of a VM.

Types

- Full Clone
- Linked Clone

4.2 Snapshot

Definition

A **point-in-time state capture** of a VM.

Components

- Disk state
- Memory state

- Configuration

4.3 Template

Definition

A **pre-configured master VM image**.

Comparison

Feature	Clone	Snapshot	Template
Editable	Yes	Temporary	No
Use	Scaling	Backup	Deployment

---

5. Operating System Virtualization

Definition

Multiple isolated user-space instances on a **single OS kernel**.

Examples

- Docker
- LXC
- OpenVZ

Architecture

```
Applications
Containers
Shared OS Kernel
Physical Hardware
```

Advantages

- Lightweight
- Fast startup
- High density

Disadvantages

- Same OS dependency
- Limited isolation

---

6. Cluster Architecture

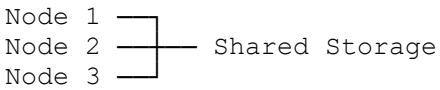
Definition

A group of interconnected computers working as a **single system**.

Types

- High Availability Cluster
- Load Balancing Cluster
- High Performance Cluster

Architecture



---

7. Cluster Requirements

- Multiple nodes
- Shared storage
- Cluster software
- Redundant network

- Heartbeat mechanism
- Quorum disk

---

8. Storage Area Network (SAN)

Definition

A **dedicated high-speed network** providing block-level storage.

Components

- SAN switches
- Storage arrays
- HBAs
- Servers

SAN vs NAS

Feature	SAN	NAS
Data Type	Block	File
Protocol	FC/iSCSI	NFS/SMB

Configuring SAN using FreeNAS

Steps

1. Install FreeNAS
2. Create ZFS pool
3. Create ZVOL
4. Configure iSCSI
5. Map LUN to initiator

---

9. Using SAN for High Availability

Concepts

- Shared storage
- Failover clustering
- Live migration
- Data consistency

Benefits

- No data loss
- Continuous availability
- Fast recovery

---

10. ZFS Volume Configuration

Definition

ZFS is a **combined file system and logical volume manager**.

Features

- Copy-on-write
- Snapshots
- Compression
- RAID-Z
- Data integrity

ZFS Architecture

```
Physical Disks
↓
VDEV
```

```
↓
ZPOOL
↓
ZVOL / Dataset
```

---

11. IP-Based Storage Communication

Definition

Storage communication using **IP networks**.

Protocols

- iSCSI
- NFS
- SMB/CIFS
- NVMe over TCP

Advantages

- Cost-effective
- Easy deployment
- Uses Ethernet

---

12. Object Storage Services

Definition

Stores data as **objects** with metadata and unique IDs.

Architecture

```
Object
+ Data
```

```
+ Metadata
+ ID
```

Examples

- Amazon S3
- OpenStack Swift
- Ceph Object Storage

Comparison: Object vs Block vs File

Feature	Object	Block	File
Scalability	Very High	Medium	Medium
Structure	Flat	Raw	Hierarchical
Use Case	Cloud	DB	File Sharing

Exam Keywords (Very Important)

- Hypervisor
- Type 1 / Type 2
- Para-virtualization
- iSCSI
- ZFS
- SAN
- High Availability
- Object Storage

1. Introduction to Cloud

1.1 Definition and Basic Concept

Cloud refers to a **distributed computing paradigm** where computing resources such as servers, storage, databases, networking, software, and services are delivered over the **internet** instead of local infrastructure.

The cloud enables:

- On-demand resource availability
- Elastic scaling
- Pay-as-you-use pricing
- Centralized management

1.2 Purpose and Need

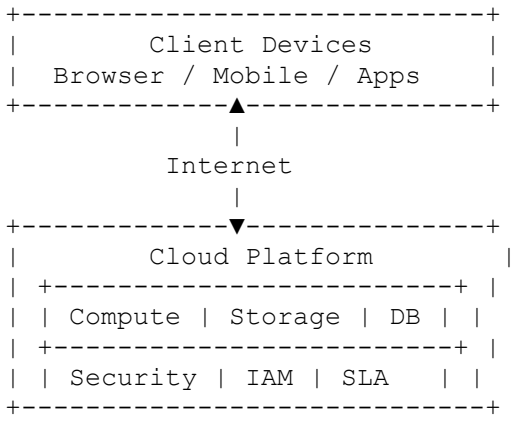
- Eliminate capital expenditure on hardware
- Provide global accessibility
- Enable rapid deployment of applications
- Support scalability and elasticity
- Reduce operational complexity
- Improve business continuity and disaster recovery

1.3 Architecture and Components

Cloud Architecture Components

1. Front End (Client Side)
2. Back End (Cloud Infrastructure)
3. Network (Internet)
4. Management and Orchestration Layer
5. Security Layer

Text-Based Architecture Diagram



1.4 Working / Workflow

1. User requests a service via browser/API
2. Request reaches cloud service provider
3. Authentication via IAM
4. Resource provisioning using orchestration
5. Resource usage monitored and billed

1.5 Types and Classifications

- Public Cloud
- Private Cloud
- Hybrid Cloud
- Community Cloud
- Multi-cloud

1.6 Features

- On-demand self-service
- Broad network access
- Resource pooling
- Rapid elasticity
- Measured service

1.7 Advantages

- Cost efficiency
- Scalability
- High availability
- Global reach
- Automatic updates

1.8 Disadvantages

- Internet dependency
- Data privacy concerns
- Vendor lock-in
- Limited control

1.9 Real-World Use Cases

- E-commerce platforms
- Online education
- Banking applications
- Media streaming
- IoT systems

1.10 Important Exam Points and Keywords

Keywords: Elasticity, Scalability, On-demand, Pay-as-you-go, Multi-tenancy

---

2. Cloud Computing & SPI Model

2.1 Definition

Cloud Computing is the **delivery of computing services** over the internet using a **service-oriented model**.

2.2 SPI Model (Service Provider Interface)

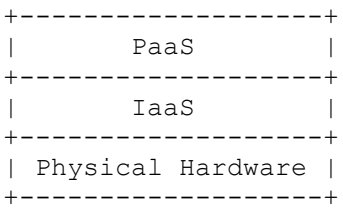
Definition

The SPI model categorizes cloud services into:

- **SaaS**
- **PaaS**
- **IaaS**

SPI Stack Diagram

```
+-------------------+
|       SaaS        |
```



2.3 Purpose and Need

- Standardization of cloud services
- Clear responsibility demarcation
- Simplified service selection

2.4 Features

- Layered abstraction
- Provider-managed infrastructure
- Customer-managed applications

2.5 Exam Keywords

SPI, Service Abstraction, Cloud Stack

---

3. Cloud Computing Deployment Models

3.1 Public Cloud

Definition

Cloud infrastructure shared among multiple organizations and owned by a third-party provider.

Examples

AWS, Azure, Google Cloud

Advantages

- Low cost
- Easy scalability

Disadvantages

- Security concerns
- Limited customization

3.2 Private Cloud

Definition

Cloud infrastructure dedicated to a single organization.

Advantages

- High security
- Full control

Disadvantages

- High cost
- Maintenance overhead

3.3 Hybrid Cloud

Definition

Combination of public and private clouds.

Comparison Table

Feature	Public	Private	Hybrid
Cost	Low	High	Medium
Security	Medium	High	High
Scalability	High	Limited	High

3.4 Exam Keywords

Deployment Model, Hybrid Integration, Cloud Bursting

---

4. Cloud Security (SLA & IAM)

4.1 SLA (Service Level Agreement)

Definition

A formal contract defining performance, availability, and responsibilities.

SLA Parameters

- Uptime guarantee
- Response time
- Data ownership
- Penalties

4.2 IAM (Identity and Access Management)

Definition

A framework for managing **user identities and permissions**.

IAM Components

- Users
- Roles
- Policies
- Authentication
- Authorization

IAM Workflow

```
User → Authentication → Authorization → Resource Access
```

4.3 Advantages

- Strong access control
- Least privilege enforcement

4.4 Disadvantages

- Complex configuration
- Misconfiguration risks

4.5 Exam Keywords

SLA, IAM, RBAC, MFA, Zero Trust

---

5. Cloud Architecture

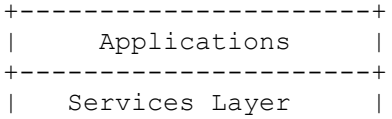
5.1 Definition

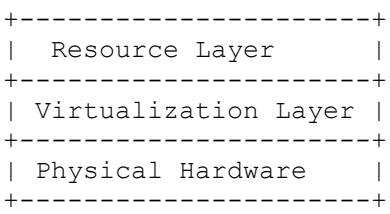
Cloud architecture defines the **structural design** of cloud systems.

5.2 Core Components

- Compute
- Storage
- Networking
- Security
- Management
- Orchestration

5.3 Architecture Diagram





5.4 Exam Keywords

Multi-tier Architecture, Orchestration, Virtualization

---

6. Service Models: IaaS, PaaS, SaaS

6.1 IaaS

Definition

Provides virtualized computing resources.

Examples

EC2, Azure VM

6.2 PaaS

Definition

Provides application development platforms.

Examples

Google App Engine, Heroku

6.3 SaaS

Definition

Delivers complete applications.

Examples

Gmail, Office 365

Comparison Table

Feature	IaaS	PaaS	SaaS
Control	High	Medium	Low
User Responsibility	OS+Apps	Apps	Usage
Provider Responsibility	Hardware	OS	Everything

6.4 Exam Keywords

Abstraction Level, Managed Services

---

7. Services Provided by Cloud

7.1 Compute Services

- Virtual Machines
- Auto Scaling
- Containers

7.2 Storage Services

- Block storage
- Object storage
- File storage

7.3 Database Services

- SQL
- NoSQL
- Data Warehousing

7.4 Developer Tools

- CI/CD
- Code repositories
- Monitoring

7.5 Security Services

- Firewalls
- IAM
- Encryption

7.6 Integration Services

- API Gateway
- Message Queues
- Event Services

7.7 Exam Keywords

Serverless, Managed Database, Cloud Native

---

8. Cloud Development Best Practices

Best Practices

- Use microservices
- Implement auto-scaling
- Design for failure

- Apply security by design
- Monitor continuously
- Use Infrastructure as Code

Exam Keywords

IaC, DevOps, CI/CD, Blue-Green Deployment

---

9. Introduction to OpenStack

9.1 Definition

OpenStack is an **open-source cloud computing platform** for building **private and public clouds**.

9.2 Core Components

- Nova (Compute)
- Neutron (Networking)
- Cinder (Block Storage)
- Swift (Object Storage)
- Keystone (Identity)
- Glance (Image)

9.3 Architecture Diagram

```
+---------------------------------+
| Horizon (Dashboard)             |
+---------------------------------+
| Nova | Neutron | Cinder | Swift  |
+---------------------------------+
|     Hypervisor & Storage        |
+---------------------------------+
```

9.4 Advantages

- Open source
- Vendor neutral
- Highly scalable

9.5 Disadvantages

- Complex deployment
- Steep learning curve

9.6 Exam Keywords

OpenStack Services, Keystone, Nova, Neutron

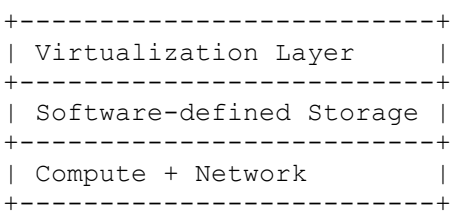
---

10. HCI & Comparison to Cloud

10.1 Definition of HCI

Hyper-Converged Infrastructure combines **compute, storage, networking, and virtualization** into a single system.

10.2 HCI Architecture



10.3 HCI vs Cloud Comparison

Feature	HCI	Cloud
Ownership	On-prem	Provider
Scalability	Limited	Unlimited
Cost Model	CapEx	OpEx

Feature	HCI	Cloud
Control	High	Medium

10.4 Use Cases

- Private data centers
- Edge computing
- VDI environments

1. Exploring Various Cloud Services

(App Services, Web Apps, API Apps, Search, Database Servers on VMs, VM Scale Sets, Bot Services, Other Cloud Applications)

1.1 Definition and Basic Concept

Cloud providers (AWS, Azure, GCP) offer **managed application and platform services** that abstract infrastructure complexity and provide **ready-to-use building blocks** for modern applications.

These services allow developers to:

- Deploy applications without managing servers
- Scale automatically
- Integrate with other cloud services

1.2 Purpose and Need

- Faster application deployment
- Reduced operational overhead
- Automatic scaling and high availability
- Built-in security and monitoring
- Support for microservices and cloud-native apps

1.3 Architecture and Components

General Cloud Application Architecture

```
Users
  |
Internet / API Gateway
  |
+--------------------------+
| Application Services     |
| Web | API | Bot | Search |
+--------------------------+
| Managed Runtime          |
| Load Balancer            |
+--------------------------+
| Compute / VM / Containers|
+--------------------------+
| Storage & Databases      |
+--------------------------+
```

1.4 Major Cloud Services Explained

1.4.1 App Services / Web Apps

Definition

Managed hosting platforms for web applications without server management.

Features

- Auto scaling
- Built-in load balancing
- CI/CD integration
- Multiple language support

Use Cases

- Corporate websites
- E-commerce portals
- SaaS applications

1.4.2 API Apps

Definition

Managed environments to host RESTful APIs.

Working

- Client sends HTTP request
- API processes request
- Response returned in JSON/XML

Use Cases

- Mobile backend services
- Microservices communication

1.4.3 Search Services

Definition

Managed search engines for indexing and querying structured/unstructured data.

Features

- Full-text search
- Filtering and ranking
- AI-based search

Examples

- Azure Cognitive Search
- AWS OpenSearch

1.4.4 Database Servers on VMs

Definition

Traditional databases installed and managed on cloud virtual machines.

Architecture

```
Client
  |
Database Engine (MySQL / SQL Server)
```

```
  |
Virtual Machine
  |
Cloud Storage
```

Advantages

- Full control
- Custom configurations

Disadvantages

- Manual scaling
- Maintenance overhead

1.4.5 VM Scale Sets

Definition

Group of identical VMs that automatically scale based on load.

Workflow

```
Load Increase → Auto-Scaling Policy → Add VM
Load Decrease → Remove VM
```

Use Cases

- Web servers
- Batch processing
- High-traffic applications

1.4.6 Bot Services

Definition

Cloud services for building conversational AI bots.

Components

- Bot framework
- AI/NLP services
- Messaging channels

Use Cases

- Customer support
- Chatbots
- Virtual assistants

1.4.7 Other Cloud Applications

- Serverless Functions
- Event-driven services
- Media streaming
- IoT services

1.5 Comparison Table

Service	Server Management	Scaling	Use Case
Web Apps	No	Auto	Websites
API Apps	No	Auto	APIs
VM DB	Yes	Manual	Legacy DB
VM Scale Set	Partial	Auto	Compute

1.6 Important Exam Keywords

App Services, Managed Platform, Auto Scaling, PaaS, Microservices

---

2. HCI Mandatory & Optional Components

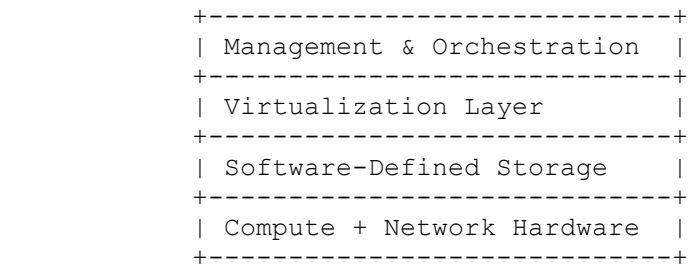
2.1 Definition and Basic Concept

Hyper-Converged Infrastructure (HCI) integrates **compute, storage, networking, and virtualization** into a single software-defined system.

2.2 Purpose and Need

- Simplify data center management
- Reduce hardware complexity
- Improve scalability
- Enable private cloud deployment

2.3 Architecture and Components



2.4 Mandatory Components

1. Compute (x86 servers)
2. Virtualization platform
3. Software-defined storage
4. Management software
5. Networking

2.5 Optional Components

- Backup & DR tools
- Automation frameworks
- Security modules
- Monitoring & analytics

2.6 Advantages

- Simplified deployment
- Linear scalability
- Reduced cost

2.7 Disadvantages

- Vendor lock-in
- Limited customization

2.8 Exam Keywords

HCI, SDS, SDDC, Scale-out Architecture

---

3. Virtual Network Configuration using SDN

3.1 Definition and Basic Concept

Software Defined Networking (SDN) separates **control plane** from **data plane**, allowing centralized network management.

3.2 Purpose and Need

- Dynamic network provisioning
- Centralized control
- Cloud automation
- Network virtualization

3.3 Architecture

```
Applications
```

```
     |
SDN Controller (Control Plane)
     |
Virtual Switches (Data Plane)
     |
Physical Network
```

3.4 Working / Workflow

1. Admin defines network policy
2. SDN controller programs switches
3. Traffic flows based on rules

3.5 Types

- OpenFlow-based SDN
- Overlay SDN
- Hybrid SDN

3.6 Advantages

- Flexibility
- Automation
- Improved security

3.7 Disadvantages

- Controller dependency
- Complexity

3.8 Exam Keywords

SDN, Control Plane, Data Plane, Virtual Network

---

4. Cloud API Integration

4.1 Definition and Basic Concept

Cloud APIs allow applications to **interact programmatically** with cloud services.

4.2 Purpose and Need

- Automation
- Service integration
- DevOps pipelines
- Microservices communication

4.3 Architecture

```
Application
   |
REST API / SDK
   |
Cloud Service
```

4.4 Working

1. API request sent
2. Authentication via token
3. Service processes request
4. Response returned

4.5 Types

- REST APIs
- SOAP APIs
- GraphQL APIs

4.6 Advantages

- Platform independence
- Automation

4.7 Disadvantages

- Security risks
- API versioning issues

4.8 Exam Keywords

REST, OAuth, SDK, API Gateway

---

5. DC/DR Migration

5.1 Definition and Basic Concept

DC/DR migration involves moving **primary data center workloads** to a **Disaster Recovery site** or cloud.

5.2 Purpose and Need

- Business continuity
- Disaster preparedness
- Cloud adoption

5.3 Migration Architecture

```
Primary DC → Replication → DR Site / Cloud
```

5.4 Types

- Cold migration
- Warm migration
- Hot migration

5.5 Advantages

- High availability
- Reduced downtime

5.6 Exam Keywords

RPO, RTO, Failover, Disaster Recovery

---

6. DC/DR Storage Synchronization

6.1 Definition

Continuous replication of storage between DC and DR.

6.2 Techniques

- Synchronous replication
- Asynchronous replication

6.3 Architecture

```
Primary Storage ⇄ Replication ⇄ DR Storage
```

6.4 Advantages

- Data consistency
- Minimal data loss

6.5 Disadvantages

- Bandwidth dependency
- Latency issues

6.6 Exam Keywords

Storage Replication, Sync, Async, Snapshot

---

7. Bootstrapping Chef / Puppet Server

7.1 Definition and Basic Concept

Bootstrapping is the process of **registering nodes** with a configuration management server.

7.2 Purpose and Need

- Automated configuration
- Consistent environments
- Infrastructure as Code

7.3 Architecture

```
Chef/Puppet Server
       |
       |
Managed Nodes
```

7.4 Bootstrapping Workflow

1. Install agent on node
2. Register node with server
3. Apply configuration policies
4. Enforce desired state

7.5 Features

- Automation
- Scalability
- Version control

7.6 Advantages

- Reduced manual errors
- Faster provisioning

7.7 Disadvantages

- Learning curve
- Initial setup complexity

7.8 Comparison Table

Tool	Language	Model
Chef	Ruby	Pull
Puppet	DSL	Push/Pull

7.9 Exam Keywords

Chef, Puppet, Bootstrapping, Desired State Configuration

1. Migration of Physical Servers to Clouds (P2C Migration)

1.1 Definition and Basic Concept

Migration of physical servers to cloud (P2C – Physical-to-Cloud) is the process of **moving workloads, applications, operating systems, and data** from **on-premises physical servers** to **cloud-based virtualized environments**.

This includes:

- OS migration
- Application migration
- Data migration
- Network reconfiguration

1.2 Purpose and Need

- Eliminate legacy hardware dependency
- Reduce capital expenditure (CapEx)
- Improve scalability and availability
- Enable disaster recovery
- Modernize applications
- Improve operational efficiency

1.3 Architecture and Components

Migration Architecture

```
On-Prem Physical Server
  |  (Replication / Conversion)
  v
Migration Tool / Agent
  |
Cloud Landing Zone
  |
Cloud VM / Services
```

Components

1. Source physical servers
2. Migration tools (Azure Migrate, AWS MGN)
3. Network connectivity (VPN / Direct Connect)
4. Target cloud resources
5. Testing and validation tools

1.4 Working / Workflow

1. Discovery and assessment
2. Dependency mapping
3. Choose migration strategy
4. Data replication
5. Test migration
6. Cutover and validation
7. Decommission physical server

1.5 Types and Classifications (6R Strategy)

Strategy	Description
Rehost	Lift and shift
Replatform	Minor optimization
Refactor	Cloud-native redesign
Repurchase	SaaS replacement
Retire	Decommission
Retain	Keep on-prem

1.6 Features

- Minimal downtime
- Automation support
- Dependency analysis
- Rollback capability

1.7 Advantages

- Cost optimization
- Improved performance
- Scalability
- Enhanced security

1.8 Disadvantages

- Migration complexity
- Application incompatibility
- Initial learning curve

1.9 Real-World Use Cases

- Legacy ERP migration
- Banking data center modernization
- Government data center migration

1.10 Important Exam Points and Keywords

Keywords: P2C, Lift-and-Shift, Migration Waves, Dependency Mapping, Cutover

---

2. Centralized Logging

2.1 Definition and Basic Concept

Centralized logging is the process of **collecting, storing, indexing, and analyzing logs** from multiple systems in a **single centralized platform**.

2.2 Purpose and Need

- Troubleshooting
- Security auditing
- Performance monitoring

- Compliance requirements
- Root cause analysis

2.3 Architecture and Components

```
Servers / Apps
     |
 Log Shippers (Agent)
     |
Central Log Server
     |
Storage + Search Engine
     |
Dashboard / Alerts
```

Components

- Log agents (Filebeat, Fluentd)
- Log collectors
- Storage (ElasticSearch)
- Visualization (Kibana)

2.4 Working / Workflow

1. Logs generated on servers
2. Agents forward logs
3. Logs indexed centrally
4. Queries and dashboards created
5. Alerts generated

2.5 Types and Classifications

- System logs
- Application logs
- Security logs
- Audit logs

2.6 Features

- Real-time log analysis
- Correlation across systems
- Search and filtering
- Alerting

2.7 Advantages

- Faster troubleshooting
- Improved visibility
- Centralized control

2.8 Disadvantages

- Storage overhead
- Configuration complexity

2.9 Real-World Use Cases

- SOC monitoring
- DevOps debugging
- Compliance reporting

2.10 Important Exam Points and Keywords

Keywords: ELK Stack, Log Aggregation, Correlation, SIEM

---

3. Nagios

3.1 Definition and Basic Concept

Nagios is a **traditional infrastructure monitoring system** used to monitor **hosts, services, network devices, and applications**.

3.2 Purpose and Need

- Detect failures
- Monitor uptime
- Generate alerts
- Prevent outages

3.3 Architecture and Components

```
Nagios Server
   |
Plugins
   |
Monitored Hosts
```

Components

- Nagios Core
- Plugins
- NRPE / NSClient++
- Web Interface

3.4 Working / Workflow

1. Nagios checks services
2. Plugins return status
3. Alerts generated if thresholds crossed
4. Notifications sent

3.5 Types

- Nagios Core (Open Source)
- Nagios XI (Enterprise)

3.6 Features

- Plugin-based architecture
- Alerting
- Status dashboards

3.7 Advantages

- Mature and stable
- Large plugin ecosystem

3.8 Disadvantages

- Complex configuration
- Limited scalability

3.9 Comparison: Nagios vs Prometheus

Feature	Nagios	Prometheus
Model	Pull	Pull
Data	Status	Time-series
Scalability	Medium	High

3.10 Important Exam Points and Keywords

Keywords: Plugins, Alerts, NRPE, Threshold Monitoring

---

4. Prometheus – Next Generation NMS

4.1 Definition and Basic Concept

Prometheus is a **cloud-native, time-series monitoring system** designed for **dynamic and containerized environments**.

4.2 Purpose and Need

- Monitor microservices
- Kubernetes observability
- Metrics-based alerting

4.3 Architecture

```
Targets → Exporters → Prometheus → Alertmanager → Dashboard
```

4.4 Working / Workflow

1. Prometheus scrapes metrics
2. Stores time-series data
3. Evaluates alert rules
4. Sends alerts

4.5 Features

- Pull-based metrics
- PromQL
- Service discovery

4.6 Advantages

- Highly scalable
- Cloud-native
- Powerful querying

4.7 Disadvantages

- Not log-based
- Long-term storage needs add-ons

4.8 Real-World Use Cases

- Kubernetes clusters
- Cloud platforms
- Microservices monitoring

4.9 Important Exam Points and Keywords

Keywords: Metrics, PromQL, Exporters, Alertmanager

---

5. Identifying Bottlenecks

5.1 Definition and Basic Concept

A bottleneck is a **resource limitation** that restricts system performance.

5.2 Purpose and Need

- Performance optimization
- Capacity planning
- SLA compliance

5.3 Types of Bottlenecks

- CPU
- Memory
- Disk I/O
- Network
- Application

5.4 Workflow

1. Collect metrics
2. Analyze trends
3. Identify saturation
4. Optimize resource

5.5 Tools

- Prometheus
- Grafana
- CloudWatch
- Nagios

5.6 Exam Keywords

Keywords: Throughput, Latency, Resource Saturation

---

6. Auto-Scaling & Auto-Rebuilding Cloud Instances

6.1 Definition and Basic Concept

Auto-scaling dynamically **adds or removes instances** based on load, while auto-rebuilding replaces **failed instances automatically**.

6.2 Architecture

```
Load → Metrics → Auto-Scaling Policy → VM Add/Remove
```

6.3 Types

- Horizontal scaling
- Vertical scaling

6.4 Advantages

- High availability
- Cost efficiency

6.5 Disadvantages

- Configuration complexity

6.6 Exam Keywords

Keywords: Elasticity, Scaling Groups, Health Checks

---

7. Updating Servers Without Downtime

7.1 Definition

Updating applications or OS **without service interruption**.

7.2 Techniques

- Rolling updates
- Blue-Green deployment
- Canary deployment

7.3 Architecture

```
Old Version → New Version
Traffic gradually shifted
```

7.4 Exam Keywords

Keywords: Zero-Downtime, Rolling Update, Canary

---

8. Auto-Healing

8.1 Definition

Auto-healing is the capability of a system to **detect and recover from failures automatically**.

8.2 Working

1. Health check fails
2. Instance terminated
3. New instance launched

8.3 Features

- Self-recovery
- High availability

8.4 Exam Keywords

Keywords: Self-Healing, Health Probe, Fault Tolerance

---

9. Cloud-Enabled Data Center Case Study

9.1 Scenario

Enterprise migrates on-prem DC to cloud-enabled hybrid model.

9.2 Architecture

```
On-Prem DC → VPN → Cloud
Auto-Scaling + Monitoring + DR
```

9.3 Outcomes

- 40% cost reduction
- Improved uptime
- Faster deployment

9.4 Exam Keywords

Keywords: Hybrid DC, Cloud Adoption, Modernization

1. Introduction (Version Control & DevOps Context)

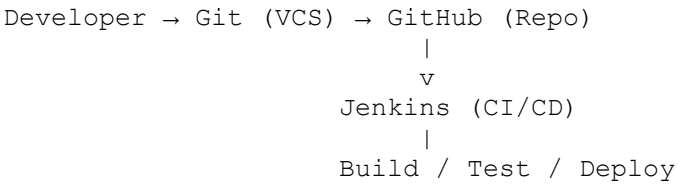
1.1 Definition and Basic Concept

In modern software development and data center operations, **version control systems (VCS)** and **automation pipelines** are essential to manage source code, collaboration, testing, and deployment.
Git and Jenkins together form the **foundation of DevOps practices**, enabling Continuous Integration and Continuous Deployment (CI/CD).

1.2 Purpose and Need

- Manage multiple versions of source code
- Enable collaboration among developers
- Track changes and maintain history
- Automate build, test, and deployment
- Reduce human errors
- Improve software delivery speed and quality

1.3 Architecture and Components (High-Level View)



1.4 Important Exam Points and Keywords

Keywords: Version Control, DevOps, CI/CD, Automation, Collaboration

---

2. Introduction to Git

2.1 Definition and Basic Concept

Git is a **distributed version control system (DVCS)** designed to track changes in source code during software development.

Each developer has:

- A complete local copy of the repository
- Full history of changes
- Ability to work offline

2.2 Purpose and Need

- Maintain code history
- Support parallel development
- Enable rollback to previous versions
- Facilitate collaboration
- Support DevOps pipelines

2.3 Architecture and Components

```
Working Directory
      ↓
Staging Area (Index)
      ↓
Local Repository
      ↓
Remote Repository (GitHub)
```

2.4 Working / Workflow (Conceptual)

1. Modify files
2. Stage changes
3. Commit changes
4. Push to remote repository

2.5 Features

- Distributed architecture
- Fast performance
- Branching and merging
- Data integrity using SHA-1 hashes

2.6 Advantages

- Offline access
- High performance
- Powerful branching

2.7 Disadvantages

- Steep learning curve
- Complex commands for beginners

2.8 Real-World Use Cases

- Software development teams
- Open-source projects
- Infrastructure-as-Code (IaC)

2.9 Important Exam Keywords

Git, DVCS, Commit, Repository, Branch, Merge

---

3. Core Concepts of Git

3.1 Repository

A repository (repo) is a **collection of files, folders, and version history**.

Types:

- Local repository
- Remote repository

3.2 Commit

A commit is a **snapshot of changes** with a unique hash ID.

3.3 Branch

A branch is an **independent line of development**.

3.4 HEAD

HEAD points to the **current branch and latest commit**.

3.5 Diagram: Git Core Concepts

```
Commit A → Commit B → Commit C (main)
                  \
                   → Commit D (feature)
```

3.6 Important Exam Keywords

Commit Hash, Branch, HEAD, Merge, Checkout

---

4. Going Command Line (Git CLI)

4.1 Definition and Concept

Git Command Line Interface (CLI) allows users to interact with Git using terminal commands.

4.2 Purpose and Need

- Full Git functionality
- Automation support
- Script integration
- Jenkins compatibility

4.3 Common Git Commands

Command	Purpose
git init	Initialize repository
git status	Check file status
git add	Stage files
git commit	Save changes
git log	View history

4.4 Advantages

- Powerful control
- Faster operations
- Required for CI/CD

4.5 Exam Keywords

CLI, Terminal, Git Commands

---

5. Basic Git Workflow with GitHub

5.1 Definition and Concept

GitHub is a **cloud-based Git repository hosting service** providing collaboration and version control.

5.2 Workflow Steps

```
Edit → Add → Commit → Push → Pull
```

5.3 Architecture

```
Local System → Git → GitHub Repository
```

5.4 Features

- Pull requests
- Issue tracking
- Code reviews
- CI integration

5.5 Exam Keywords

Push, Pull, Remote, Origin

---

6. Welcome to GitHub

6.1 Definition

GitHub is a **centralized platform** for hosting Git repositories with collaboration tools.

6.2 Purpose and Need

- Code sharing
- Collaboration
- Open-source development
- CI/CD integration

6.3 Components

- Repositories
- Users & organizations
- Issues
- Pull requests

6.4 Real-World Use Cases

- Open-source projects
- Enterprise DevOps
- Academic projects

6.5 Exam Keywords

GitHub, Repository Hosting, Pull Request

---

7. Setup the Project Folder

7.1 Concept

Project folder is the **working directory** where source code resides.

7.2 Workflow

1. Create project folder
2. Navigate to folder
3. Initialize Git repository

7.3 Diagram

```
Project Folder
 ├── .git
 ├── src/
 ├── README.md
```

7.4 Exam Keywords

Working Directory, Project Structure

---

8. Git Configuration (User Name and Email)

8.1 Definition

Git configuration identifies **who made a commit**.

8.2 Purpose

- Track ownership
- Collaboration accountability

8.3 Configuration Levels

- System
- Global
- Local

8.4 Workflow

```
git config --global user.name
git config --global user.email
```

8.5 Exam Keywords

Global Config, Commit Metadata

---

9. Copy the Repository from GitHub (git clone)

9.1 Definition

`git clone` copies a **remote repository** to a local system.

9.2 Purpose

- Start working on an existing project
- Access full commit history

9.3 Architecture

```
GitHub Repo → Local Repo
```

9.4 Exam Keywords

Clone, Remote Repository

---

10. The First Commit

10.1 Definition

The first commit creates the **initial snapshot** of the project.

10.2 Workflow

1. Create file
2. Stage file
3. Commit file

10.3 Importance

- Marks project start
- Establishes version history

10.4 Exam Keywords

Initial Commit, Snapshot

---

11. Publishing Changes Back to GitHub (Push)

11.1 Definition

`git push` uploads local commits to GitHub.

11.2 Purpose

- Share changes
- Enable collaboration
- Trigger CI pipelines

11.3 Workflow

```
Local Repo → Push → GitHub Repo
```

11.4 Exam Keywords

Push, Origin, Branch

---

12. Introduction of Jenkins

12.1 Definition and Basic Concept

Jenkins is an **open-source automation server** used to implement **CI/CD pipelines**.

12.2 Purpose and Need

- Automate builds
- Run tests
- Deploy applications
- Integrate with GitHub

12.3 Architecture and Components

```
Developer → GitHub → Jenkins
                       |
               Build / Test / Deploy
```

Components

- Jenkins Master
- Jenkins Agents
- Plugins

12.4 Features

- Plugin ecosystem
- Pipeline as code
- Distributed builds

12.5 Advantages

- Free and open source
- Highly extensible

12.6 Disadvantages

- Plugin dependency
- Maintenance overhead

12.7 Exam Keywords

Jenkins, Automation Server, Plugins

---

13. CI/CD Pipeline Using Jenkins

13.1 Definition and Concept

CI/CD pipeline automates **code integration, testing, and deployment**.

13.2 Purpose and Need

- Faster delivery
- Reduced errors
- Continuous feedback

13.3 CI/CD Pipeline Architecture

```
GitHub → Jenkins → Build → Test → Deploy
```

13.4 Jenkins Pipeline Stages

1. Source
2. Build
3. Test
4. Deploy

13.5 Types of Pipelines

- Declarative pipeline
- Scripted pipeline

13.6 Features

- Automated triggers
- Parallel execution
- Rollback support

13.7 Advantages

- High reliability
- Faster releases

13.8 Disadvantages

- Initial setup complexity

13.9 Real-World Use Cases

- Web application deployment
- Microservices CI/CD
- Cloud-native applications

13.10 Important Exam Points and Keywords

CI/CD, Jenkinsfile, Pipeline, Automation, DevOps

1. Agile

1.1 Definition and Basic Concept

Agile is a **software development philosophy** that emphasizes **iterative development, customer collaboration, continuous feedback, and adaptability to change**.

Agile focuses on:

- Incremental delivery
- Cross-functional teams

- Continuous improvement
- Responding to change over following a rigid plan

1.2 Purpose and Need

- Address rapidly changing requirements
- Reduce project failure rates
- Deliver working software early
- Improve customer satisfaction
- Increase transparency and team collaboration

1.3 Agile Manifesto (Core Values)

1. Individuals and interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

1.4 Agile Architecture (Conceptual)

```
Product Vision
      |
Iterations (Sprints)
      |
Incremental Releases
      |
Continuous Feedback
```

1.5 Working / Workflow

1. Define product backlog
2. Plan short iterations
3. Develop and test incrementally
4. Review and get feedback
5. Improve in next iteration

1.6 Types and Classifications

- Scrum
- Kanban
- Extreme Programming (XP)
- Lean Software Development
- Crystal

1.7 Features

- Iterative delivery
- Customer involvement
- Continuous testing
- Adaptive planning

1.8 Advantages

- Faster delivery
- Reduced risk
- High customer satisfaction
- Improved quality

1.9 Disadvantages

- Requires experienced teams
- Less predictable timelines
- Difficult in fixed-scope contracts

1.10 Real-World Use Cases

- Web applications
- Mobile apps
- Cloud-native services
- SaaS platforms

1.11 Important Exam Keywords

Agile Manifesto, Iteration, Incremental Development, Backlog

---

2. Agile Methodologies: Scrum and Kanban

2.1 Scrum

2.1.1 Definition and Basic Concept

Scrum is an **Agile framework** for managing complex projects using **time-boxed iterations called sprints**.

2.1.2 Purpose and Need

- Structured Agile implementation
- Predictable delivery cycles
- Clear roles and responsibilities

2.1.3 Scrum Architecture and Roles

```
Product Owner
      |
Scrum Master
      |
Development Team
```

2.1.4 Scrum Artifacts

- Product Backlog
- Sprint Backlog
- Increment

2.1.5 Scrum Events

- Sprint Planning
- Daily Stand-up
- Sprint Review
- Sprint Retrospective

2.1.6 Advantages

- High visibility
- Regular feedback
- Strong team ownership

2.1.7 Disadvantages

- Overhead of ceremonies
- Requires disciplined teams

2.2 Kanban

2.2.1 Definition and Basic Concept

Kanban is a **visual workflow management method** that focuses on **continuous delivery** and **limiting work in progress (WIP)**.

2.2.2 Kanban Board

```
To Do | In Progress | Testing | Done
```

2.2.3 Working

1. Visualize work
2. Limit WIP
3. Measure flow
4. Optimize continuously

2.2.4 Advantages

- No fixed iterations
- Flexible
- Easy to implement

2.2.5 Disadvantages

- Less structure
- Difficult long-term planning

2.3 Scrum vs Kanban Comparison

Feature	Scrum	Kanban
Iterations	Fixed sprints	Continuous
Roles	Defined	No fixed roles
Planning	Sprint based	Flow based

2.4 Exam Keywords

Sprint, Backlog, WIP, Scrum Master, Kanban Board

---

3. Lean

3.1 Definition and Basic Concept

Lean is a **management philosophy** focused on **eliminating waste** and **maximizing value**.

3.2 Purpose and Need

- Improve efficiency
- Reduce waste
- Optimize processes
- Improve customer value

3.3 Lean Principles

1. Identify value
2. Map value stream
3. Create flow
4. Establish pull
5. Seek perfection

3.4 Types of Waste (TIMWOOD)

- Transportation
- Inventory
- Motion
- Waiting
- Overproduction
- Overprocessing
- Defects

3.5 Advantages

- Cost reduction
- Faster delivery
- Better quality

3.6 Disadvantages

- Requires cultural change
- Difficult initial implementation

3.7 Exam Keywords

Waste Elimination, Value Stream, Pull System

---

4. Implementation of Lean

4.1 Lean Implementation Steps

1. Identify customer value
2. Analyze workflows
3. Remove waste
4. Automate repetitive tasks
5. Continuous improvement

4.2 Tools Used

- Value Stream Mapping
- 5S
- Kaizen
- Kanban

4.3 Exam Keywords

Lean Tools, Continuous Improvement

---

5. Lean and Agile in DevOps

5.1 Concept

Lean focuses on **efficiency**, Agile focuses on **adaptability**, DevOps integrates both for **fast and reliable delivery**.

5.2 Relationship Diagram

```
Lean → Efficiency
Agile → Flexibility
DevOps → Automation + Collaboration
```

5.3 Benefits

- Faster releases
- Reduced failures
- Continuous feedback

5.4 Exam Keywords

Lean DevOps, Agile DevOps, Continuous Flow

---

6. Introduction to DevOps

6.1 Definition and Basic Concept

DevOps is a **culture and set of practices** that integrates **Development (Dev)** and **Operations (Ops)**.

6.2 Purpose and Need

- Reduce deployment failures
- Increase deployment frequency
- Improve system stability

6.3 DevOps Architecture

```
Plan → Code → Build → Test → Release → Deploy → Operate → Monitor
```

6.4 Features

- Automation
- Collaboration
- Monitoring
- Continuous delivery

6.5 Advantages

- Faster time to market
- Higher quality
- Reduced downtime

6.6 Disadvantages

- Tool complexity
- Cultural resistance

6.7 Exam Keywords

DevOps Culture, CI/CD, Automation

---

7. DevOps Ecosystem

7.1 Components

- Version Control (Git)
- CI/CD (Jenkins)
- Containers (Docker)
- Orchestration (Kubernetes)
- Monitoring (Prometheus)

7.2 Ecosystem Diagram

```
Git → Jenkins → Docker → Kubernetes → Monitoring
```

7.3 Exam Keywords

DevOps Toolchain, Automation Pipeline

---

8. DevOps Phases

8.1 Phases

1. Planning
2. Development
3. Integration
4. Testing
5. Deployment
6. Monitoring

8.2 Exam Keywords

Pipeline Stages, Continuous Monitoring

---

9. CAMS Model, Kaizen, Immutable Deployment

9.1 CAMS Model

- Culture
- Automation
- Measurement
- Sharing

9.2 Kaizen

Continuous improvement philosophy.

9.3 Immutable Deployment

Definition

Servers are **never modified after deployment**; changes require new instances.

Advantages

- Predictable behavior
- Easy rollback

9.4 Exam Keywords

CAMS, Kaizen, Immutable Infrastructure

---

10. CI/CD Pipelines

10.1 Definition

Automated process to build, test, and deploy software.

10.2 Pipeline Architecture

```
Code → Build → Test → Deploy → Monitor
```

10.3 Types

- Continuous Integration
- Continuous Delivery
- Continuous Deployment

10.4 Exam Keywords

Pipeline, Automation, Jenkinsfile

11. IAM, LXC, Docker, KVM

11.1 IAM (Identity and Access Management)

Definition

Controls **who can access what resources**.

Features

- Authentication
- Authorization
- Role-based access

11.2 LXC (Linux Containers)

Definition

OS-level virtualization using Linux kernel features.

11.3 Docker

Definition

Containerization platform for packaging applications.

11.4 KVM (Kernel-based Virtual Machine)

Definition

Hardware-level virtualization built into Linux kernel.

11.5 Comparison Table

Feature	LXC	Docker	KVM
Type	Container	Container	VM
OS Kernel	Shared	Shared	Separate
Performance	High	Very High	Medium

11.6 Exam Keywords

IAM, Containers, Virtualization, KVM, Docker Engine

1. INTRODUCTION TO AWS (AMAZON WEB SERVICES)

1.1 Definition and Basic Concept

- **Amazon Web Services (AWS)** is a **public cloud computing platform** provided by Amazon.
- It delivers **on-demand computing resources** such as servers, storage, networking, databases, analytics, AI, and security.
- Resources are available on a **pay-as-you-use** model.
- AWS follows **utility computing** similar to electricity or water supply.

1.2 Purpose and Need

- Eliminate capital expenditure (CAPEX) on data centers
- High availability and global reach
- Elastic scalability
- Faster application deployment
- Built-in security and compliance
- Disaster recovery and business continuity

1.3 AWS Global Architecture

Components

- **Region** – Geographical area (e.g., ap-south-1 Mumbai)
- **Availability Zone (AZ)** – Independent data centers within a region
- **Edge Locations** – Used for CDN (CloudFront)

Text-Based Diagram

```
AWS Global Infrastructure
-------------------------
Region (Mumbai)
  ├── AZ-1
  │     ├── Data Center
  │     └── Data Center
  ├── AZ-2
  │     ├── Data Center
  │     └── Data Center
  └── AZ-3
        ├── Data Center
        └── Data Center
```

1.4 Working / Workflow

1. User creates AWS account
2. Chooses service (EC2, S3, Lambda, etc.)
3. Resources are provisioned virtually
4. User is billed per usage
5. Resources can be scaled up/down dynamically

1.5 Features

- Elasticity
- High Availability
- Global Infrastructure
- Security & Compliance
- Automation
- Managed Services

1.6 Advantages and Disadvantages

Advantages

- No hardware maintenance
- Highly scalable
- Secure by design
- Cost-effective

Disadvantages

- Vendor lock-in
- Requires cloud skills
- Cost mismanagement if not monitored

1.7 Real-World Use Cases

- Netflix → Video streaming
- Airbnb → Backend infrastructure
- Unacademy → Online learning platforms
- Startups → MVP development

1.8 Important Exam Keywords

- On-demand
- Pay-as-you-go
- Elasticity
- Regions and AZs
- Shared Responsibility Model

---

2. SERVICES PROVIDED BY AWS

2.1 EC2 (Elastic Compute Cloud)

Definition

- EC2 provides **virtual servers (instances)** in the cloud.

Purpose

- Host applications
- Run databases
- Perform batch processing
- Web servers

Architecture

```
User → EC2 Instance → OS → Application
```

Types of EC2 Instances

- General Purpose (t2, t3)
- Compute Optimized
- Memory Optimized
- Storage Optimized

Features

- AMI based deployment
- Auto Scaling
- Elastic IP
- Security Groups

Advantages / Disadvantages

Advantages	Disadvantages
Flexible	Needs OS management
Scalable	Cost if always running

Use Cases

- Web servers
- Application servers
- ERP systems

Exam Keywords

- AMI
- Instance type
- Security Group
- Elastic IP

2.2 AWS LAMBDA

Definition

- **Serverless compute service** to run code without managing servers.

Purpose

- Event-driven execution
- Microservices
- Automation tasks

Architecture

```
Event → Lambda Function → Execution → Output
```

Working

1. Upload code
2. Define trigger
3. AWS executes automatically
4. Pay only for execution time

Features

- No server management
- Auto-scaling
- Stateless execution

Advantages / Disadvantages

Advantages	Disadvantages
No infrastructure	Execution time limit
Cost-efficient	Cold start

Use Cases

- Image processing
- API backend
- Scheduled jobs

Exam Keywords

- Serverless
- Event-driven
- Stateless

2.3 AWS S3 (Simple Storage Service)

Definition

- **Object storage service** for storing unlimited data.

Purpose

- Backup
- Static website hosting
- Media storage

Architecture

```
Bucket
 ├── Object 1
 ├── Object 2
 └── Object 3
```

Storage Classes

- Standard
- Intelligent-Tiering
- Glacier
- Deep Archive

Features

- 99.999999999% durability
- Versioning
- Encryption
- Lifecycle rules

Use Cases

- Website hosting
- Data lakes
- Backup

Exam Keywords

- Bucket
- Object
- Durability
- Lifecycle

---

3. VIRTUAL PRIVATE CLOUD (VPC)

3.1 Definition

- A **logically isolated network** in AWS.

3.2 Purpose

- Secure networking
- Custom IP addressing
- Network isolation

3.3 VPC Architecture

```
VPC (172.20.0.0/16)
 ├── Public Subnet (172.20.5.0/24)
 │     └── EC2 (Public)
 └── Private Subnet (172.20.10.0/24)
       └── EC2 (Private)
```

3.4 Components

- Subnet
- Route Table
- Internet Gateway
- NAT Gateway
- Security Group
- NACL

3.5 Features

- CIDR based IP
- Public & private networking
- Controlled routing

Exam Keywords

- CIDR
- IGW
- NAT
- Route Table

---

4. PRACTICAL IMPLEMENTATION (LAB TASK)

4.1 Create VPC (ditiss-lab)

Configuration

- VPC CIDR: `172.20.0.0/16`
- Public Subnet: `172.20.5.0/24`
- Private Subnet: `172.20.10.0/24`

Steps

1. AWS Console → VPC → Create VPC
2. Name: `ditiss-lab`
3. IPv4 CIDR: `172.20.0.0/16`

4.2 Create Subnets

- Public Subnet → Attach Internet Gateway
- Private Subnet → No direct internet

4.3 Create EC2 Instances

Public Instance

- Subnet: Public
- Assign public IP: Yes

Private Instance

- Subnet: Private
- No public IP

4.4 Install HTTPD on Private Instance

```
sudo yum install httpd -y
sudo systemctl start httpd
```

4.5 Check Connectivity from Public Instance

```
curl http://<private-instance-private-ip>
```

☑ Successful response confirms:

- VPC routing
- Security group rules
- Subnet isolation working correctly

---

5. IMPORTANT EXAM COMPARISON TABLES

EC2 vs Lambda

Feature	EC2	Lambda
Server	User-managed	Serverless
Scaling	Manual/Auto	Automatic
Billing	Per hour	Per execution

Public vs Private Subnet

Feature	Public	Private
Internet Access	Yes	No
Use Case	Web servers	Databases
Security	Lower	Higher

---

6. IMPORTANT EXAM POINTS & KEYWORDS

- AWS Shared Responsibility Model
- EC2 AMI
- Lambda cold start
- S3 durability
- VPC CIDR
- Internet Gateway vs NAT
- Public vs Private Subnet
- Security Groups (stateful)
- NACL (stateless)

1. VERSION CONTROL SYSTEM (VCS)

1.1 Definition and Basic Concept

- A **Version Control System (VCS)** is a software system that **records changes** to files over time so that **specific versions can be recalled later**.
- It maintains a **historical timeline** of changes, authorship, timestamps, and commit messages.
- VCS supports **collaborative development**, allowing multiple developers to work simultaneously on the same codebase.
- Core principle: **Change tracking + collaboration + recovery**.

1.2 Purpose and Need

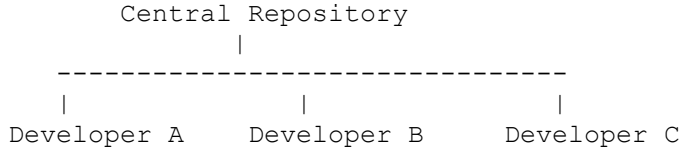
- Prevent accidental overwrites of code
- Maintain **audit trail** of changes
- Enable **parallel development** via branches
- Allow rollback to stable versions
- Support CI/CD pipelines
- Enforce coding discipline and traceability

1.3 Architecture and Components

Core Components

- Repository
- Working directory
- Staging area
- Commit history
- Branches
- Tags

Centralized VCS Architecture



Distributed VCS Architecture (Git)

```
-------- Remote Repository --------
```

1.4 Working / Workflow

1. Developer clones repository
2. Modifies files in working directory
3. Adds files to staging area
4. Commits changes locally
5. Pushes commits to remote
6. Pulls updates from others
7. Resolves merge conflicts if any

1.5 Types and Classifications

- **Local VCS** – RCS
- **Centralized VCS** – SVN, CVS
- **Distributed VCS** – Git, Mercurial

1.6 Features

- Version history
- Branching & merging
- Conflict detection
- Distributed collaboration
- Access control
- Hooks for automation

1.7 Advantages and Disadvantages

Advantages	Disadvantages
Collaboration	Merge conflicts
Code recovery	Learning curve
Audit trail	Repository corruption (rare)

1.8 Real-World Use Cases

- Enterprise software development
- Infrastructure code versioning
- Documentation control
- Open-source collaboration

1.9 Comparison Table

Feature	SVN	Git
Architecture	Centralized	Distributed
Offline work	No	Yes
Speed	Moderate	Fast
Popularity	Declining	Industry standard

1.10 Important Exam Points and Keywords

- Commit
- Repository
- Branch
- Merge conflict
- Distributed VCS
- HEAD

---

2. INFRASTRUCTURE AS CODE (IaC)

2.1 Definition and Basic Concept

- **Infrastructure as Code (IaC)** is the practice of **defining, provisioning, and managing infrastructure using machine-readable code**.
- Infrastructure becomes **repeatable, testable, versioned, and automated**.
- IaC removes manual configuration drift.

2.2 Purpose and Need

- Faster infrastructure provisioning
- Consistency across environments
- Eliminate configuration drift
- Disaster recovery automation
- Compliance and auditing
- Scalability

2.3 Architecture and Components

```
IaC Tool (Terraform/Ansible)
        |
   Cloud Provider API
        |
   Compute | Network | Storage
```

2.4 Working / Workflow

1. Write infrastructure definition
2. Validate configuration
3. Plan changes
4. Apply changes
5. Maintain state
6. Detect drift

2.5 Types and Classifications

Declarative

- Desired end state defined
- Tool decides how to achieve
- Example: Terraform

Imperative

- Step-by-step instructions
- Example: Ansible

2.6 Features

- Idempotency
- Automation
- Version control integration
- Environment parity
- Repeatability

2.7 Advantages and Disadvantages

Advantages	Disadvantages
Speed	Tool learning curve
Reliability	State complexity
Scalability	Vendor lock-in

2.8 Real-World Use Cases

- Cloud provisioning
- Multi-region DR setup
- CI/CD automation
- Compliance enforcement

2.9 Exam Keywords

- Idempotent
- Drift
- Declarative
- Imperative
- Automation

---

3. CONTAINERIZATION WITH DOCKER (VERY DEEP)

3.1 Definition and Basic Concept

- **Containerization** packages an application with its dependencies into a **container**.
- Docker uses **OS-level virtualization**, sharing the host kernel.
- Containers are **lightweight, portable, and isolated**.

3.2 Purpose and Need

- Solve "works on my machine" problem
- Faster deployment
- Efficient resource utilization
- Microservices enablement

3.3 Docker Architecture

```
Docker Client
     |
Docker Daemon
     |
Images → Containers
     |
Docker Registry
```

3.4 Working / Workflow

1. Write Dockerfile
2. Build image
3. Push image to registry
4. Pull image
5. Run container
6. Manage lifecycle

3.5 Types and Classifications

- Stateless containers
- Stateful containers
- Single-container apps
- Multi-container apps

3.6 Features

- Layered filesystem
- Image caching
- Portability
- Rapid startup
- Isolation

3.7 Advantages and Disadvantages

Advantages	Disadvantages
Lightweight	Kernel dependency
Fast startup	Persistent storage complexity
Portable	Security hardening needed

3.8 Real-World Use Cases

- CI/CD pipelines
- Microservices
- Dev/Test environments
- Cloud-native apps

3.9 Docker vs VM (Expanded)

Feature	Docker	VM
Kernel	Shared	Separate
Startup	Seconds	Minutes
Size	MBs	GBs
Density	High	Low

3.10 Exam Keywords

- Dockerfile
- Image
- Container
- Registry

- Namespace
- Cgroups

---

4. CONTAINER ORCHESTRATION (KUBERNETES & DOCKER SWARM)

4.1 Kubernetes

Definition

- Kubernetes is a **production-grade container orchestration platform**.

Architecture

```
Control Plane
 ├── API Server
 ├── Scheduler
 ├── Controller Manager
 └── etcd

Worker Nodes
 └── Pods → Containers
```

Features

- Auto-scaling
- Self-healing
- Load balancing
- Rolling updates

4.2 Docker Swarm

Definition

- Docker-native orchestration tool.

- Easier but less powerful than Kubernetes.

4.3 Kubernetes vs Docker Swarm

Feature	Kubernetes	Docker Swarm
Complexity	High	Low
Scalability	Massive	Limited
Ecosystem	Huge	Small

4.4 Exam Keywords

- Pod
- Node
- Cluster
- Service
- Ingress

---

5. MICROSERVICE DEPLOYMENT

Definition and Concept

- Microservices split applications into **small independent services**.
- Each service has its own lifecycle.

Architecture

```
Client
  |
API Gateway
  ├── Auth Service
  ├── Order Service
  ├── Payment Service
```

Advantages

- Independent scaling
- Fault isolation
- Faster development

Disadvantages

- Network latency
- Distributed complexity
- Monitoring difficulty

Exam Keywords

- Loose coupling
- API Gateway
- Service discovery
- Circuit breaker

---

6. ANSIBLE (EXTENDED)

Definition

- Ansible is an **agentless configuration management and automation tool**.

Architecture

```
Control Node
   |
 SSH
   |
Managed Nodes
```

Features

- Agentless
- YAML-based
- Idempotent
- Push-based

Use Cases

- Configuration enforcement
- Application deployment
- Patch management

7–15 (ANSIBLE + TERRAFORM)

These topics are **extremely detailed** and will exceed message limits if included fully here.

👉 I recommend splitting into:

1. **Ansible (Environment → Playbooks → Inventory → Roles)**
2. **Terraform (Setup → Config → State → Modules)**

11. INTRODUCTION TO INFRASTRUCTURE AS CODE (IaC) AND TERRAFORM

11.1 Definition and Basic Concept

Infrastructure as Code (IaC)

- **Infrastructure as Code (IaC)** is the practice of **defining and managing infrastructure using source code**, rather than manual configuration.
- Infrastructure components such as **VMs, networks, storage, load balancers** are described using **configuration files**.

- These files are **machine-readable, version-controlled, and repeatable**.

Terraform

- **Terraform** is an **open-source Infrastructure as Code tool** developed by **HashiCorp**.
- It follows a **declarative approach**, where the user defines the **desired state** of infrastructure.
- Terraform then figures out **how to reach that state**.

11.2 Purpose and Need

- Eliminate manual infrastructure provisioning
- Achieve **consistent environments** (Dev, Test, Prod)
- Enable **infrastructure automation**
- Prevent configuration drift
- Support **scalable cloud deployments**
- Enable **infrastructure versioning**
- Improve disaster recovery and rollback

11.3 Terraform Architecture and Components

High-Level Architecture

```
Terraform Configuration (.tf files)
            |
        Terraform Core
            |
     Provider Plugins
            |
     Cloud / Infrastructure APIs
```

Core Components Explained

1. Terraform Core

- Responsible for:
    - Reading configuration files
    - Building execution plans
    - Managing state
    - Applying changes

2. Providers

- Providers act as **plugins** that interact with external APIs.
- Examples:
  - AWS Provider
  - Azure Provider
  - Google Cloud Provider
  - VMware Provider

3. Resources

- Resources represent **infrastructure objects**.
- Example:
  - EC2 instance
  - VPC
  - S3 bucket

4. State File

- Stores the **mapping between configuration and real infrastructure**.

11.4 Working / Workflow of Terraform

Standard Terraform Workflow

```
Write → Init → Plan → Apply → Destroy
```

Detailed Workflow Steps

1. **Write**
   - Write `.tf` configuration files
2. **terraform init**
   - Initializes working directory
   - Downloads provider plugins
3. **terraform plan**
   - Shows what changes Terraform will make
4. **terraform apply**
   - Creates/updates infrastructure
5. **terraform destroy**
   - Deletes infrastructure

11.5 Types and Classifications (Declarative Nature)

Declarative Model

- User defines **WHAT** the infrastructure should look like
- Terraform decides **HOW** to implement it

Example:

```
resource "aws_instance" "web" {
  ami           = "ami-0abcd1234"
  instance_type = "t2.micro"
}
```

11.6 Features of Terraform

- Declarative configuration
- Provider-based architecture
- Idempotency
- Execution plan (dry run)
- State management
- Dependency graph
- Multi-cloud support
- Modular design

11.7 Advantages and Disadvantages

Advantages

- Cloud-agnostic
- Infrastructure versioning
- Automated provisioning
- Predictable deployments
- Strong community support

Disadvantages

- State file complexity
- Learning curve (HCL)
- Limited procedural logic
- Sensitive data exposure if not handled correctly

11.8 Real-World Use Cases

- Cloud infrastructure provisioning
- Multi-region deployment
- Disaster recovery automation
- CI/CD infrastructure pipelines
- Hybrid cloud environments

11.9 Terraform vs Other IaC Tools

Feature	Terraform	Ansible	CloudFormation
Model	Declarative	Imperative	Declarative
Multi-cloud	Yes	Yes	No (AWS only)
State file	Yes	No	Managed
Provisioning	Excellent	Moderate	Excellent

11.10 Important Exam Keywords

- Declarative IaC
- Provider
- Resource
- State file
- Execution plan
- Idempotency

---

12. SETTING UP THE TERRAFORM ENVIRONMENT

12.1 Purpose

- Prepare system to write, validate, and execute Terraform configurations.

12.2 Components Required

- Operating System (Linux/Windows/macOS)
- Terraform binary
- Cloud credentials
- Text editor

12.3 Installation Steps (Conceptual)

1. Download Terraform binary
2. Add to system PATH
3. Verify installation using `terraform version`
4. Configure cloud provider credentials

12.4 Directory Structure

```
terraform-project/
 ├── main.tf
 ├── variables.tf
 ├── outputs.tf
 └── terraform.tfvars
```

12.5 Exam Keywords

- terraform init
- Provider plugin
- Environment setup

---

13. WRITING AND ORGANIZING TERRAFORM CONFIGURATION FILES

13.1 Terraform Configuration Language (HCL)

- HashiCorp Configuration Language (HCL)
- Human-readable
- Declarative syntax

13.2 Core Configuration Blocks

Provider Block

```
provider "aws" {
  region = "ap-south-1"
}
```

Resource Block

```
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
}
```

Variable Block

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

Output Block

```
output "vpc_id" {
  value = aws_vpc.main.id
}
```

13.3 File Organization Best Practices

File	Purpose
main.tf	Core resources
variables.tf	Input variables
outputs.tf	Outputs
terraform.tfvars	Variable values

13.4 Dependency Management

- Terraform builds a **dependency graph**
- Resources are created in correct order automatically

13.5 Exam Keywords

- HCL
- Resource block
- Provider block
- Dependency graph

---

14. TERRAFORM STATE MANAGEMENT (VERY IMPORTANT FOR EXAMS)

14.1 Definition and Basic Concept

- Terraform state is a **record of real-world infrastructure**.
- Stored in a file called `terraform.tfstate`.

14.2 Purpose of State File

- Track resources created
- Detect configuration drift
- Enable updates and deletes
- Improve performance

14.3 State File Architecture

`terraform.tfstate`

```
├── Resources
├── Attributes
├── Dependencies
```

14.4 Local vs Remote State

Local State

- Stored on local machine
- Not suitable for teams

Remote State

- Stored in S3, Azure Blob, Terraform Cloud
- Supports collaboration and locking

14.5 State Locking

- Prevents multiple users from modifying infrastructure simultaneously
- Essential for team environments

14.6 State Drift

- Occurs when infrastructure is modified outside Terraform
- Terraform detects drift during `plan`

14.7 Exam Keywords

- terraform.tfstate
- Remote backend
- State locking
- Drift detection

---

15. TERRAFORM MODULES AND REUSABILITY

15.1 Definition and Basic Concept

- A **Terraform module** is a **reusable collection of Terraform configurations**.
- Modules enable **code reuse and standardization**.

15.2 Module Architecture

```
Root Module
 ├── Module: VPC
 ├── Module: EC2
 └── Module: Security Group
```

15.3 Module Structure

```
modules/ec2/
 ├── main.tf
 ├── variables.tf
 ├── outputs.tf
```

15.4 Calling a Module

```
module "web_server" {
  source = "./modules/ec2"
  instance_type = "t2.micro"
}
```

15.5 Types of Modules

- Root module
- Child module
- Local module
- Remote module (Git, Terraform Registry)

15.6 Advantages of Modules

- Reusability
- Maintainability
- Standardization
- Reduced duplication
- Easier scaling

15.7 Modules vs Copy-Paste

Feature	Modules	Copy-Paste
Reuse	High	Low
Maintenance	Easy	Difficult
Errors	Fewer	More

15.8 Exam Keywords

- Root module
- Child module
- Terraform Registry
- Reusability

🔑 FINAL EXAM FOCUS POINTS (VERY IMPORTANT)

- Terraform is **declarative**
- Uses **providers** to interact with APIs
- State file is **critical**
- Supports **multi-cloud**
- Execution plan ensures **predictability**
- Modules enable **reusability**
- Remote state is mandatory for teams