Cloud & Data Testing Foundations

Docker, Kubernetes, Big Data & ETL Testing

MODULE 1

What is Cloud Computing?

Cloud computing is the delivery of computing services—including servers, storage, databases, networking, software, analytics, and intelligence—over the internet ("the cloud") to offer faster innovation, flexible resources, and economies of scale. Rather than owning their own computing infrastructure or data centres, organisations can rent access to anything from applications to storage from a cloud service provider.

On-Demand Resources

Access computing power instantly without hardware investment

Pay-as-You-Go

Pay only for cloud services you use, reducing operating costs

Global Scale

Deliver services from geographically distributed data centres worldwide

MODULE 1

Cloud Service Models: IaaS vs PaaS vs SaaS

Understanding the three primary cloud service models is essential for determining testing strategies and responsibilities. Each model offers different levels of control, flexibility, and management requirements.

Infrastructure as a Service (IaaS)

Provides virtualised computing resources over the internet. You manage the operating systems, applications, and data whilst the provider manages servers, storage, and networking.

- Examples: AWS EC2, Azure Virtual Machines, Google Compute Engine
- Testing focus: Infrastructure provisioning, network configuration, VM performance

Platform as a Service (PaaS)

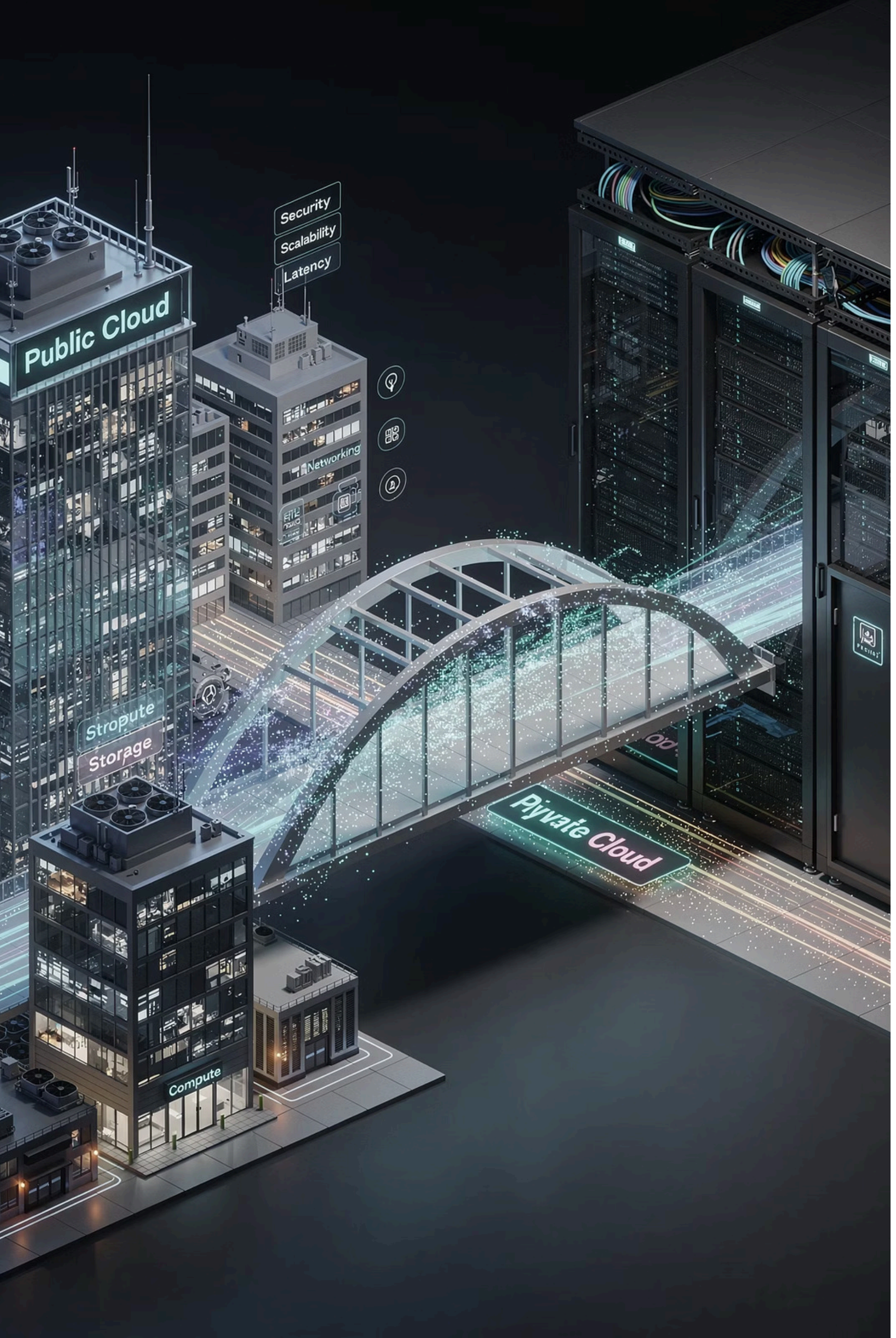
Provides a platform allowing customers to develop, run, and manage applications without dealing with infrastructure. The provider manages everything except your application and data.

- Examples: AWS Elastic Beanstalk, Azure App Service, Google App Engine
- Testing focus: Application functionality, integration points, platform compatibility

Software as a Service (SaaS)

Provides complete software applications over the internet. The provider manages everything including the application itself.

- Examples: Salesforce, Microsoft 365, Google Workspace, Dropbox
- Testing focus: User acceptance, business workflows, integration with other systems



MODULE 1

Cloud Deployment Models

Public Cloud

Services delivered over the public internet and shared across organisations. Resources are owned and operated by third-party providers.

- Cost-effective and scalable
- No maintenance overhead
- Examples: AWS, Azure, GCP

Private Cloud

Computing resources used exclusively by one organisation. Can be hosted on-premises or by a third-party provider.

- Enhanced security and control
- Customisable infrastructure
- Higher costs and maintenance

Hybrid Cloud: Combines public and private clouds, allowing data and applications to be shared between them. Provides greater flexibility, optimisation of existing infrastructure, and more deployment options whilst maintaining security for sensitive data.

MODULE 1

Why Cloud Testing is Needed

Traditional testing approaches are insufficient for cloud-based applications due to their distributed nature, dynamic scaling, and multi-tenant architecture. Cloud testing ensures applications perform reliably across different environments, handle varying loads, and maintain security in shared infrastructures.

Scalability Validation

Verify applications can handle increased loads by adding resources. Test both horizontal scaling (adding instances) and vertical scaling (increasing instance capacity).

Elasticity Testing

Ensure systems automatically scale up during peak demand and scale down during low usage. Verify cost efficiency through proper resource allocation.

Multi-Tenancy Security

Validate data isolation between tenants sharing the same infrastructure. Ensure one tenant cannot access another tenant's data or resources.

Distributed Systems

Test communication between geographically dispersed components. Verify data consistency, network latency, and failover mechanisms across regions.

MODULE 1

Challenges in Cloud Testing

Testing cloud applications presents unique challenges that differ significantly from traditional testing environments. Understanding these challenges enables testers to develop appropriate strategies and mitigation approaches.

Key Challenges

Limited Control

Testers have restricted access to underlying infrastructure and cannot directly monitor hardware or network components.

Environment Complexity

Multiple environments, regions, and configurations create complex test scenarios requiring extensive coverage.

Data Security

Testing with production-like data whilst maintaining compliance with privacy regulations poses significant challenges.

Network Dependency

Internet connectivity issues can affect test execution and results, requiring robust retry and timeout mechanisms.

Mitigation Strategies

- Leverage cloud provider monitoring and logging services for visibility
- Implement infrastructure as code (IaC) for consistent environment provisioning
- Use data masking and synthetic data generation for secure testing
- Design tests with network resilience patterns including retries and circuit breakers
- Employ chaos engineering to validate system behaviour under failure conditions
- Establish clear SLAs with cloud providers for performance baselines

MODULE 1

Cloud Testing vs Traditional Testing

Aspect	Traditional Testing	Cloud Testing
Infrastructure	Fixed, on-premises hardware	Dynamic, virtualised resources
Scalability	Limited by physical capacity	Virtually unlimited, on-demand scaling
Environment Setup	Time-consuming, manual provisioning	Rapid, automated provisioning
Cost Model	Capital expenditure (CapEx)	Operational expenditure (OpEx), pay-per-use
Maintenance	In-house team manages hardware	Provider handles infrastructure maintenance
Geographic Testing	Requires physical presence or VPN	Test from multiple regions easily
Disaster Recovery	Complex backup and restore procedures	Built-in redundancy and failover capabilities
Test Data Management	Local databases and storage	Distributed data stores, multi-region replication

MODULE 1

Cloud Architecture for Testing

A typical cloud architecture comprises multiple layers designed for high availability, scalability, and fault tolerance. Testing must validate each layer independently and ensure proper integration between components.

ASSESSMENT

Cloud Testing Knowledge Check

Test your understanding of cloud computing fundamentals and testing concepts covered in this module.

Question 1

Which cloud service model provides the most control over the underlying infrastructure?

- A) SaaS
- B) PaaS
- C) IaaS ✓
- D) FaaS

Explanation: IaaS provides virtualised computing resources where users manage the OS, applications, and data whilst the provider manages physical infrastructure.

Question 2

What is the primary difference between scalability and elasticity in cloud computing?

- A) Scalability is manual, elasticity is automatic ✓
- B) They are the same concept
- C) Scalability is for storage, elasticity is for compute
- D) Elasticity only applies to private clouds

Explanation: Scalability refers to the ability to increase capacity (manually or automatically), whilst elasticity specifically means automatic scaling up and down based on demand.

HANDS-ON EXERCISE

Exercise 1: Cloud Testing Strategy

Scenario: Your organisation is migrating a customer-facing e-commerce web application to AWS. The application experiences variable traffic with peak loads during seasonal sales. You need to create a comprehensive cloud testing strategy.

01

Identify Test Types

Determine which test types are critical for this cloud deployment: performance testing, security testing, scalability testing, disaster recovery testing, and multi-region failover testing.

02

Create Test Strategy Outline

Document your approach including: test objectives, test environments (dev, staging, production-like), tools required (JMeter, Selenium, AWS CloudWatch), and success criteria for each test type.

03

Environment Validation Checklist

Develop a checklist covering: network connectivity, security groups configuration, load balancer setup, database connectivity, monitoring and alerting configuration, and backup mechanisms.

Discussion Points: Consider how you would simulate peak loads, validate auto-scaling behaviour, test failover scenarios, and ensure data security during testing. Document any assumptions and constraints.

MODULE 2

What is Containerisation?

Containerisation is a lightweight form of virtualisation that packages an application and all its dependencies, libraries, and configuration files into a single, portable unit called a container. Unlike traditional virtualisation, containers share the host operating system's kernel, making them more efficient and faster to start.

Containers ensure that software runs consistently regardless of where it's deployed—whether on a developer's laptop, a test environment, or production servers. This "build once, run anywhere" approach eliminates the classic "it works on my machine" problem and significantly streamlines the development, testing, and deployment pipeline.

Consistency

Identical environments across development, testing, and production

Isolation

Applications run independently without interfering with each other

Portability

Run containers on any infrastructure supporting the container runtime

MODULE 2

Containers vs Virtual Machines

Understanding the fundamental differences between containers and virtual machines is crucial for making informed decisions about application deployment and testing strategies. Both technologies provide isolation, but they achieve it through different mechanisms with distinct trade-offs.

Characteristic	Virtual Machines	Containers
Architecture	Includes full OS with hypervisor	Share host OS kernel
Size	Gigabytes (typically 10-100 GB)	Megabytes (typically 10-500 MB)
Startup Time	Minutes	Seconds
Resource Usage	Higher overhead	Lightweight, minimal overhead
Isolation Level	Complete isolation with separate OS	Process-level isolation
Performance	Slower due to hypervisor layer	Near-native performance
Portability	Limited, OS-dependent	Highly portable across platforms
Use Case	Running different OS, strong isolation	Microservices, rapid deployment, scaling

MODULE 2

Container Architecture Comparison

The architectural difference is fundamental: VMs virtualise the hardware layer and include complete operating systems, whilst containers virtualise the operating system and share the kernel, resulting in dramatically reduced overhead and faster startup times.

MODULE 2

Why Containers Matter for Testing

Containers have revolutionised software testing by providing consistent, reproducible environments that eliminate configuration drift and environmental inconsistencies. Testers can now create isolated test environments in seconds, run parallel tests without interference, and ensure test results are reliable and repeatable.

The containerisation approach aligns perfectly with modern CI/CD practices, enabling automated testing pipelines where each test run occurs in a fresh, identical environment. This consistency dramatically reduces false positives and environment-related test failures.

Environment Parity

Test in production-like environments without complex setup procedures

Rapid Provisioning

Spin up test environments in seconds rather than hours or days

Resource Efficiency

Run multiple isolated test environments on a single host machine

Version Control

Store container configurations as code in version control systems

Clean Testing

Each test starts with a clean slate, eliminating residual state issues

ASSESSMENT

Containerisation Knowledge Check

Assess your understanding of containerisation concepts and their advantages over traditional virtualisation.

Question 1

What is the primary reason containers start faster than virtual machines?

- A) Containers use more powerful hardware
- B) Containers share the host OS kernel ✓
- C) Containers have fewer applications
- D) Containers don't require networking

Explanation: Containers share the host operating system's kernel and don't need to boot a complete OS, resulting in startup times measured in seconds rather than minutes.

Question 2

Which scenario is most appropriate for using containers rather than VMs?

- A) Running Windows applications on Linux
- B) Microservices requiring rapid scaling ✓
- C) Complete isolation between untrusted workloads
- D) Running different operating system kernels simultaneously

Explanation: Containers excel at microservices architectures where rapid deployment, scaling, and efficient resource usage are priorities. VMs are better for running different operating systems or when maximum isolation is required.

MODULE 3

What is Docker?

Docker is the industry-leading containerisation platform that enables developers to package applications into standardised units called containers. Introduced in 2013, Docker revolutionised software development and deployment by making containerisation accessible, efficient, and reliable.

Docker provides a complete ecosystem for building, sharing, and running containers. It includes tools for creating container images, a runtime for executing containers, and a registry for distributing container images. Docker's popularity stems from its simplicity, extensive documentation, and vast ecosystem of pre-built images.

> "Docker containers wrap a piece of software in a complete filesystem that contains everything needed to run: code, runtime, system tools, and libraries. This guarantees that the software will always run the same, regardless of its environment."

MODULE 3

Docker Architecture

Docker uses a client-server architecture where the Docker client communicates with the Docker daemon, which performs the heavy lifting of building, running, and distributing containers. Understanding these components is essential for effective Docker usage.

Core Components

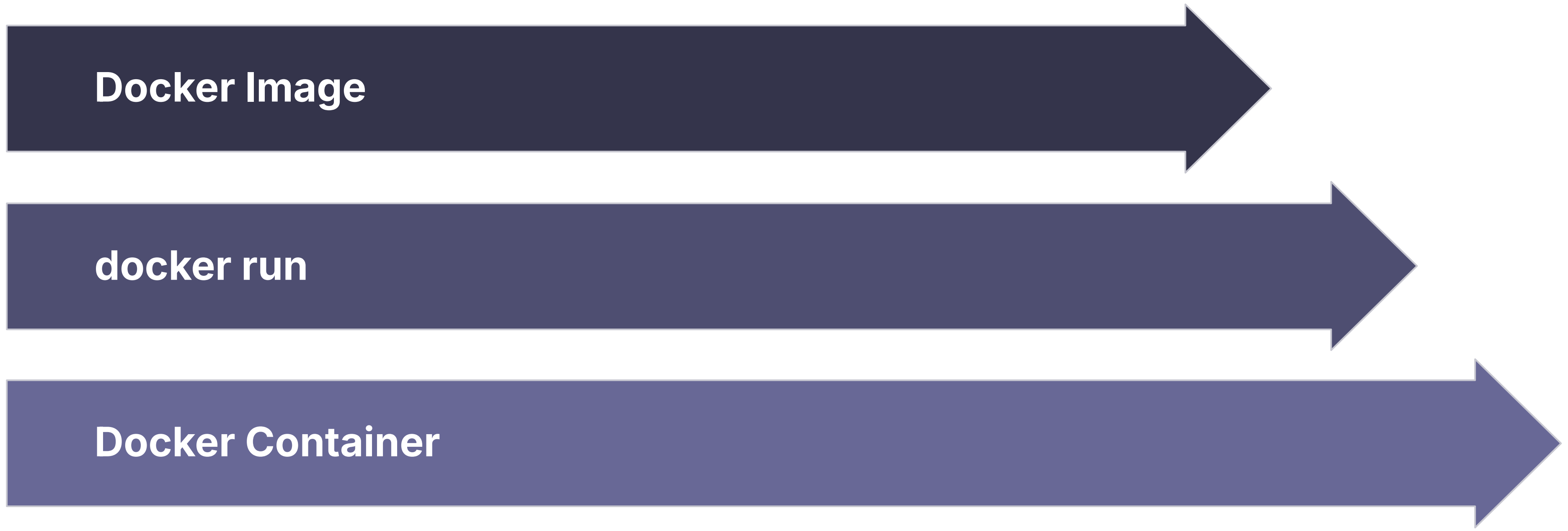
- **Docker Client:** Command-line interface that users interact with using docker commands
- **Docker Daemon (dockerd):** Background service managing Docker objects like images, containers, networks, and volumes
- **Docker Registry:** Repository for storing and distributing Docker images (Docker Hub is the public registry)

Docker Objects

- **Images:** Read-only templates containing instructions for creating containers
- **Containers:** Runnable instances of images with their own isolated filesystem and processes
- **Networks:** Enable communication between containers
- **Volumes:** Persist data generated by containers

MODULE 3

Docker Images vs Containers



MODULE 3

Understanding Dockerfile

A Dockerfile is a text document containing instructions for building a Docker image. Each instruction creates a layer in the image, enabling efficient caching and reuse. Writing effective Dockerfiles is crucial for creating optimised, secure container images.

Sample Dockerfile

```
# Base image
FROM node:18-alpine

# Set working directory
WORKDIR /app

# Copy package files
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy application code
COPY . .

# Expose port
EXPOSE 3000

# Define startup command
CMD ["npm", "start"]
```

Common Dockerfile Instructions

- **FROM:** Specifies the base image to build upon
- **WORKDIR:** Sets the working directory inside the container
- **COPY/ADD:** Copies files from host to container
- **RUN:** Executes commands during image build
- **EXPOSE:** Documents which ports the container listens on
- **ENV:** Sets environment variables
- **CMD:** Specifies the default command to run
- **ENTRYPOINT:** Configures container as an executable

MODULE 3

Docker Image Layers

Docker images are built using a layered architecture where each instruction in a Dockerfile creates a new layer. These layers are cached and reused, making builds faster and storage more efficient. Understanding layers is crucial for optimising image size and build times.

Layer Caching

Unchanged layers are reused from cache, significantly speeding up builds

Layer Sharing

Multiple images sharing the same base layers save storage space

Build Optimisation

Order Dockerfile instructions to maximise cache efficiency

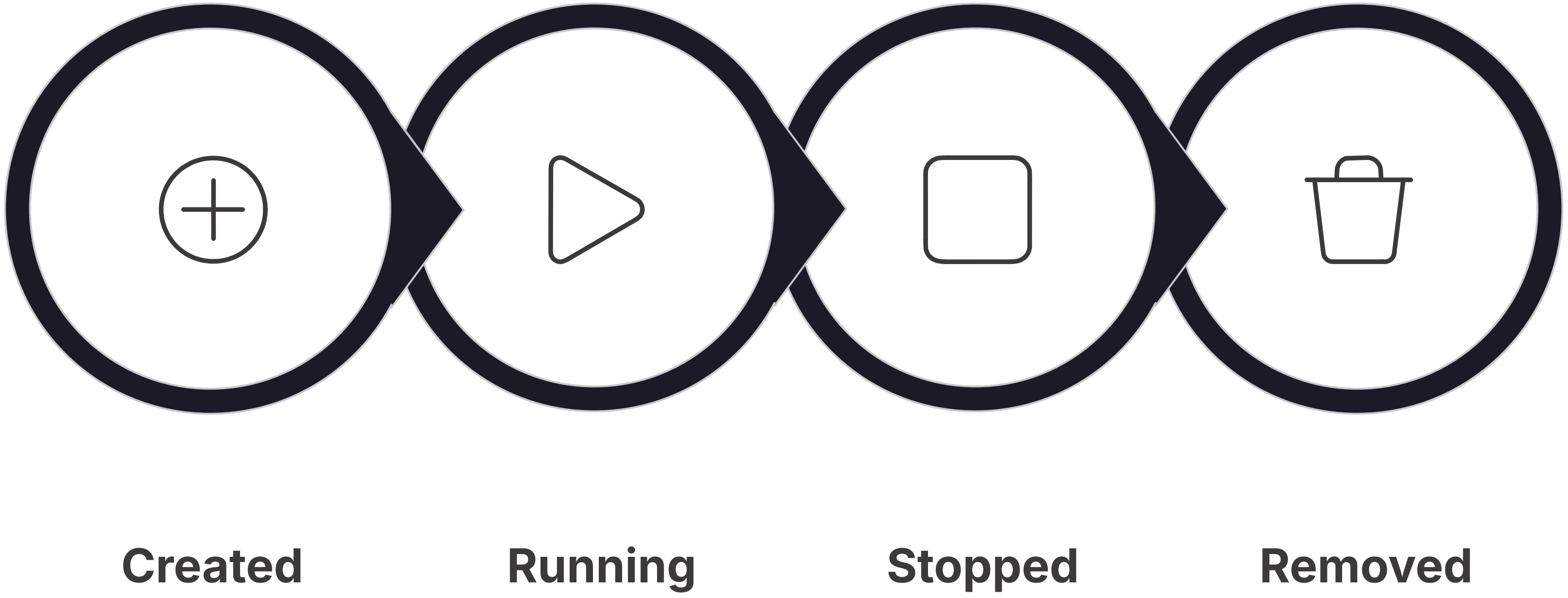
Writable Layer

Containers add a thin writable layer on top of read-only image layers

MODULE 3

Docker Container Lifecycle

Containers transition through various states from creation to termination. Understanding this lifecycle is essential for managing containers effectively and troubleshooting issues.



MODULE 3

Essential Docker Commands: Version & Images

Mastering Docker commands is fundamental to working effectively with containers. Let's start with commands for checking your Docker installation and managing images.

docker --version

Purpose: Displays the installed Docker version

```
docker --version
# Output: Docker version 24.0.6, build ed223bc
```

Use this command to verify Docker is installed correctly and check which version you're running.

docker pull

Purpose: Downloads an image from a registry (Docker Hub by default)

```
docker pull nginx:latest
docker pull ubuntu:22.04
docker pull postgres:15-alpine
```

Images are specified as `name:tag`. If no tag is provided, Docker uses `latest` by default. Alpine variants are smaller, optimised images.

docker images

Purpose: Lists all images stored locally on your system

```
docker images
# Shows: REPOSITORY, TAG, IMAGE ID, CREATED, SIZE
```

Also available as `docker image ls`. Use `-a` flag to show intermediate layers and `-q` to display only image IDs.

MODULE 3

Essential Docker Commands: Running Containers

The `docker run` command is one of the most important Docker commands. It creates and starts a new container from an image with extensive configuration options.

docker run (Basic)

Purpose: Creates and starts a new container from an image

```
docker run nginx
docker run -d nginx        # Detached mode (background)
docker run -it ubuntu bash   # Interactive with terminal
```

Common flags: `-d` (detached), `-it` (interactive terminal), `--name` (assign name), `--rm` (auto-remove after exit)

docker run (Port Mapping)

Purpose: Maps container ports to host ports for external access

```
docker run -d -p 8080:80 nginx
docker run -d -p 5432:5432 postgres:15
```

Format: `-p host_port:container_port`. The first port is accessible on your host machine, the second is the port inside the container.

docker run (Environment Variables)

Purpose: Passes configuration to containers via environment variables

```
docker run -e POSTGRES_PASSWORD=secret postgres
docker run -e NODE_ENV=production -e PORT=3000 myapp
```

Use `-e` or `--env` to set individual variables, or `--env-file` to load variables from a file.

MODULE 3

Essential Docker Commands: Managing Containers

Once containers are running, you need commands to monitor, control, and clean them up. These management commands are essential for day-to-day Docker operations.

Viewing Containers

```
# List running containers
docker ps

# List all containers (including stopped)
docker ps -a

# Show only container IDs
docker ps -q
```

The `docker ps` command displays container ID, image, command, status, ports, and names. Understanding this output is crucial for container management.

Stopping Containers

```
# Stop a running container gracefully
docker stop container_name

# Stop using container ID
docker stop a3f9b8c2d1e0

# Stop multiple containers
docker stop web_app db_server
```

Docker sends SIGTERM, waits for graceful shutdown (default 10 seconds), then sends SIGKILL if necessary. Use `docker kill` for immediate termination.

Removing Containers

```
# Remove a stopped container
docker rm container_name

# Force remove a running container
docker rm -f container_name

# Remove all stopped containers
docker container prune
```

Container Logs & Info

```
# View container logs
docker logs container_name

# Follow log output
docker logs -f container_name

# Inspect container details
docker inspect container_name
```

MODULE 3

Building Custom Docker Images

Creating custom images from Dockerfiles allows you to package your applications with all dependencies. The `docker build` command reads the Dockerfile and creates an image layer by layer.

01

Create a Dockerfile

Write a Dockerfile defining your application's environment, dependencies, and startup command. Save it as `Dockerfile` (no extension) in your project directory.

02

Build the Image

Run `docker build -t myapp:1.0 .` where `-t` tags the image with a name and version, and `.` specifies the build context (current directory).

03

Verify the Build

Use `docker images` to confirm your image appears in the list. Check the size and ensure the build completed without errors.

04

Run Your Container

Start a container from your new image: `docker run -d -p 8080:8080 myapp:1.0`. Test that your application works as expected.

Build Context: The dot (`.`) in the build command specifies the build context—the directory Docker uses to find files referenced in the Dockerfile. Large contexts slow down builds, so use `.dockerignore` to exclude unnecessary files.

MODULE 3

Docker Command Reference Summary

Command	Purpose	Example
`docker --version`	Show Docker version	`docker --version`
`docker pull`	Download image from registry	`docker pull nginx:alpine`
`docker images`	List local images	`docker images -a`
`docker run`	Create and start container	`docker run -d -p 80:80 nginx`
`docker ps`	List running containers	`docker ps -a`
`docker stop`	Stop running container	`docker stop my_container`
`docker rm`	Remove stopped container	`docker rm my_container`
`docker build`	Build image from Dockerfile	`docker build -t myapp:1.0 .`
`docker logs`	View container logs	`docker logs -f my_container`
`docker exec`	Execute command in running container	`docker exec -it my_container bash`

HANDS-ON EXERCISE

Exercise 2: Docker Practical Lab

Objective: Gain hands-on experience with Docker by pulling images, running containers, building custom images, and managing the container lifecycle. This exercise reinforces fundamental Docker operations essential for testing environments.

1

Step 1: Pull and Run Nginx

```
# Pull the nginx image
docker pull nginx:alpine

# Run nginx in detached mode with port mapping
docker run -d -p 8080:80 --name my_nginx nginx:alpine

# Verify container is running
docker ps

# Access in browser: http://localhost:8080
```

Expected Output: You should see the Nginx welcome page. The container status should show "Up" in `docker ps` output.

2

Step 2: Inspect and Manage

```
# View container logs
docker logs my_nginx

# View detailed container information
docker inspect my_nginx

# Stop the container
docker stop my_nginx

# Verify it's stopped
docker ps -a

# Remove the container
docker rm my_nginx
```

Note: After stopping, `docker ps` won't show the container, but `docker ps -a` will display it with "Exited" status.

HANDS-ON EXERCISE

Exercise 2 (Continued): Build Custom Image

01

Create Project Files

Create a directory my-web-app and inside it create two files: index.html with basic HTML content and a Dockerfile.

```
# index.html
<html>
  <body>
    <h1>My Dockerised Web App</h1>
    <p>Successfully built and running!</p>
  </body>
</html>
```

02

Write the Dockerfile

```
# Dockerfile
FROM nginx:alpine
COPY index.html /usr/share/nginx/html/
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

This Dockerfile uses nginx as the base image and copies your HTML file to nginx's web root directory.

03

Build Your Image

```
# Navigate to project directory
cd my-web-app

# Build the image with a tag
docker build -t my-custom-app:1.0 .

# Verify the image was created
docker images | grep my-custom-app
```

The build process will show each step and layer creation. Watch for any errors.

04

Run Your Custom Container

```
# Run your custom image
docker run -d -p 8080:80 --name custom_web my-custom-app:1.0

# Check it's running
docker ps

# Test in browser: http://localhost:8080
# You should see your custom HTML page

# View logs
docker logs custom_web

# Cleanup when done
docker stop custom_web && docker rm custom_web
```

HANDS-ON EXERCISE

Exercise 2: Common Issues & Troubleshooting

Understanding common Docker errors and their solutions is crucial for effective troubleshooting. Here are issues you might encounter during the lab exercises.

Port Already in Use

Error: "Bind for 0.0.0.0:8080 failed: port is already allocated"

Solution: Either stop the container using that port (`docker ps` to find it), or use a different host port: `docker run -p 8081:80 nginx`

Container Name Conflict

Error: "The container name '/my_nginx' is already in use"

Solution: Remove the existing container: `docker rm my_nginx`, or use a different name, or use `--rm` flag to auto-remove containers after they stop.

Image Not Found

Error: "Unable to find image 'nginx:latest' locally"

Solution: This is actually normal—Docker will automatically pull the image. If it fails, check your internet connection and verify the image name and tag are correct.

Permission Denied

Error: "Permission denied while trying to connect to Docker daemon socket"

Solution: Add your user to the docker group: `sudo usermod -aG docker $USER`, then log out and back in, or prefix commands with `sudo`.

MODULE 4

Why Kubernetes is Needed

Docker excels at running individual containers, but production applications typically consist of dozens or hundreds of containers distributed across multiple machines. Managing these containers manually becomes impossible at scale—you need automated deployment, scaling, load balancing, and self-healing capabilities.

Kubernetes (K8s) solves the "containers at scale" problem. It automates deployment, scaling, and management of containerised applications across clusters of machines. When a container fails, Kubernetes automatically replaces it. When traffic increases, Kubernetes adds more containers. When you need to update your application, Kubernetes performs rolling updates with zero downtime.

Automated Scaling

Dynamically scale applications up or down based on demand

Self-Healing

Automatically restart failed containers and reschedule them on healthy nodes

Load Balancing

Distribute traffic across multiple container instances automatically

Rolling Updates

Deploy updates gradually without downtime, with automatic rollback on failures

MODULE 4

The Containers at Scale Problem

Managing containerised applications manually works for small deployments but quickly becomes unmanageable as complexity grows. Consider the challenges faced by modern applications requiring high availability and scalability.

Manual Management Challenges

- **Deployment Complexity:** Manually deploying containers across multiple hosts is error-prone and time-consuming
- **Health Monitoring:** Continuously checking if containers are healthy and restarting failed ones requires constant attention
- **Resource Optimisation:** Efficiently packing containers onto available hardware to maximise utilisation is difficult
- **Network Configuration:** Setting up service discovery and load balancing between containers manually is complicated
- **Update Management:** Coordinating updates across multiple containers whilst maintaining availability is risky

Scale Requirements

100+

Containers

Modern apps run hundreds of container instances

24/7

Availability

Zero-downtime requirements for critical services

10x

Traffic Spikes

Need to handle sudden demand increases

Kubernetes addresses these challenges through automation, providing declarative configuration where you specify the desired state and Kubernetes maintains it automatically.

MODULE 4

Kubernetes Architecture Overview

Kubernetes uses a master-worker architecture where the control plane manages the cluster and worker nodes run the actual application containers. Understanding this architecture is fundamental to working with Kubernetes.

Control Plane (Master)

The brain of the Kubernetes cluster, making decisions about scheduling, detecting and responding to events, and maintaining desired state.

- **API Server:** Central management point, handles all API requests
- **Scheduler:** Assigns Pods to worker nodes based on resource requirements
- **Controller Manager:** Runs controllers that maintain desired state
- **etcd:** Distributed key-value store holding all cluster data

Worker Nodes

Machines that run containerised applications. Each node runs several Kubernetes components to enable container orchestration.

- **Kubelet:** Agent ensuring containers run in Pods as specified
- **Container Runtime:** Software running containers (Docker, containerd)
- **Kube-proxy:** Maintains network rules for Pod communication
- **Pods:** Smallest deployable units containing one or more containers

MODULE 4

Understanding Pods

A Pod is the smallest deployable unit in Kubernetes and represents a single instance of a running process in your cluster. Pods encapsulate one or more containers, storage resources, a unique network IP, and configuration about how the containers should run.

Pod Characteristics

Containers within a Pod share the same network namespace, meaning they can communicate via localhost. They also share storage volumes, making it easy to share data between containers in the same Pod.

Most commonly, Pods contain a single container. Multi-container Pods are used when containers are tightly coupled—for example, a main application container and a helper "sidecar" container providing supporting functionality like logging or monitoring.

- Pods are ephemeral—they can be created and destroyed dynamically
- Each Pod gets a unique IP address
- Containers in a Pod share storage volumes
- Pods are the unit of scaling in Kubernetes
- Kubernetes manages Pods, not individual containers

Pod Lifecycle: Pods are mortal. When a Pod dies, Kubernetes doesn't resurrect the same Pod—it creates a new one with a new IP. This is why you need Services for stable networking.

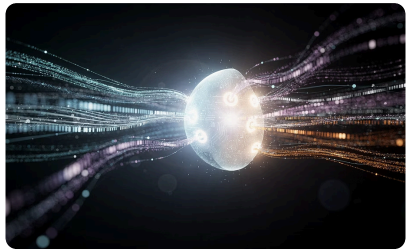
MODULE 4

Kubernetes Services

Since Pods are ephemeral and can be created or destroyed at any time, their IP addresses change frequently. Services provide a stable endpoint for accessing Pods, acting as an abstraction layer that defines a logical set of Pods and a policy for accessing them.

Load Balancing

Services automatically distribute traffic across multiple Pod replicas, ensuring no single Pod is overwhelmed. The built-in load balancing improves application reliability and performance.



Stable Networking

Services provide a consistent DNS name and IP address even as underlying Pods are replaced. Applications connect to the Service, not individual Pods, ensuring continuity.

Service Discovery

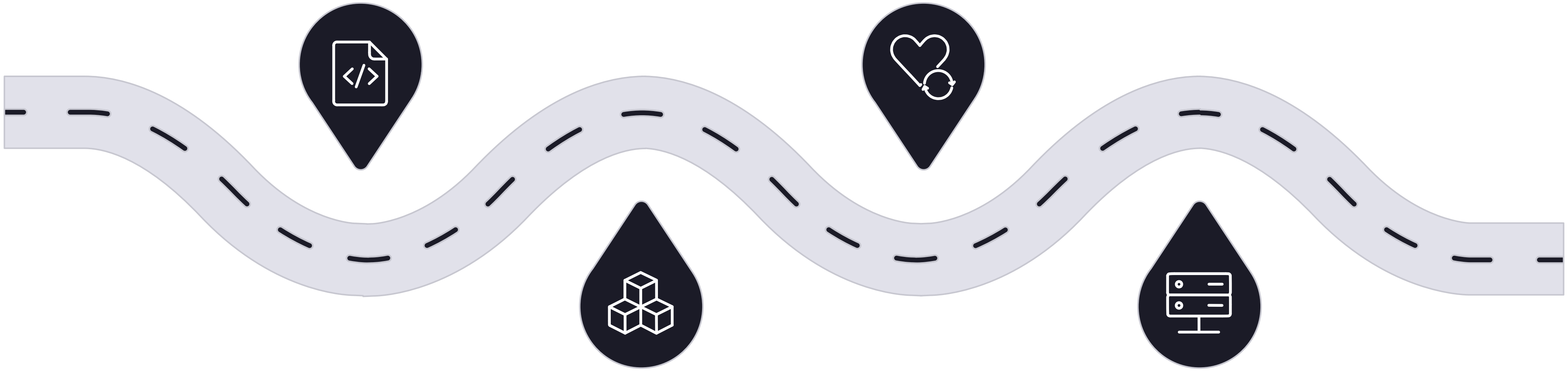
Kubernetes automatically updates Service endpoints as Pods are added or removed. Applications don't need to track individual Pod IPs—they simply use the Service name.

Service types include ClusterIP (internal cluster access), NodePort (exposes on each node's IP), LoadBalancer (exposes via cloud provider's load balancer), and ExternalName (maps to external DNS name).

MODULE 4

Kubernetes Deployments

A Deployment provides declarative updates for Pods and ReplicaSets. You describe the desired state in a Deployment manifest, and Kubernetes changes the actual state to match. Deployments are the recommended way to manage application lifecycle in Kubernetes.



Define State

Declare desired pod and replica configuration

Create Pods

Kubernetes instantiates Pods from the spec

Monitor Health

Liveness and readiness probes track status

Maintain Replicas

Controller ensures desired replica count

MODULE 4

Scaling Containers with Kubernetes

One of Kubernetes' most powerful features is its ability to scale applications dynamically. Scaling can be manual or automatic, responding to resource utilisation or custom metrics to ensure applications have sufficient capacity.

Horizontal Scaling

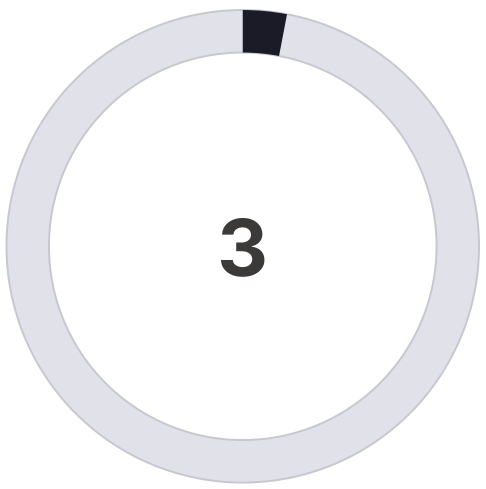
Add or remove Pod replicas to handle varying load levels. This is the most common scaling approach in Kubernetes.

```
# Manual scaling
kubectl scale deployment myapp --replicas=5

# Autoscaling based on CPU
kubectl autoscale deployment myapp \
  --min=2 --max=10 --cpu-percent=80
```

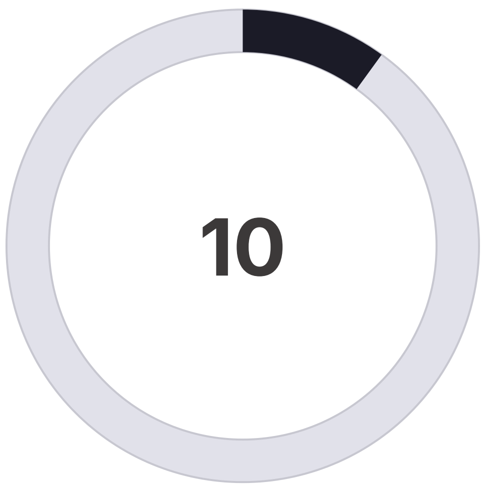
The Horizontal Pod Autoscaler (HPA) monitors metrics and automatically adjusts replica count. When CPU usage exceeds the threshold, HPA adds Pods. When usage drops, it removes Pods to save resources.

Scaling Scenarios



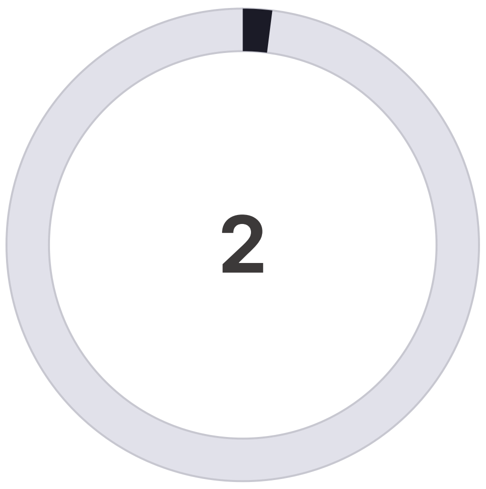
Baseline

Normal traffic, 3 replicas handle the load efficiently



Peak Traffic

During peaks, automatically scale up to 10 replicas



Off-Peak

Scale down to 2 replicas during low-traffic periods

Kubernetes continuously monitors and adjusts, ensuring applications always have adequate resources whilst minimising costs during low-demand periods.

MODULE 4

Self-Healing in Kubernetes

Kubernetes provides automatic self-healing capabilities that detect and recover from failures without human intervention. This dramatically improves application reliability and reduces operational burden.

Liveness Probes

Kubernetes periodically checks if containers are alive. If a liveness probe fails, Kubernetes kills the container and restarts it according to the Pod's restart policy.

Readiness Probes

Determines when a container is ready to accept traffic. Failed readiness probes remove the Pod from Service load balancing until it passes, preventing routing to unhealthy instances.

Node Failure Recovery

When a node becomes unhealthy or unreachable, Kubernetes automatically reschedules all Pods from that node onto healthy nodes, maintaining application availability.

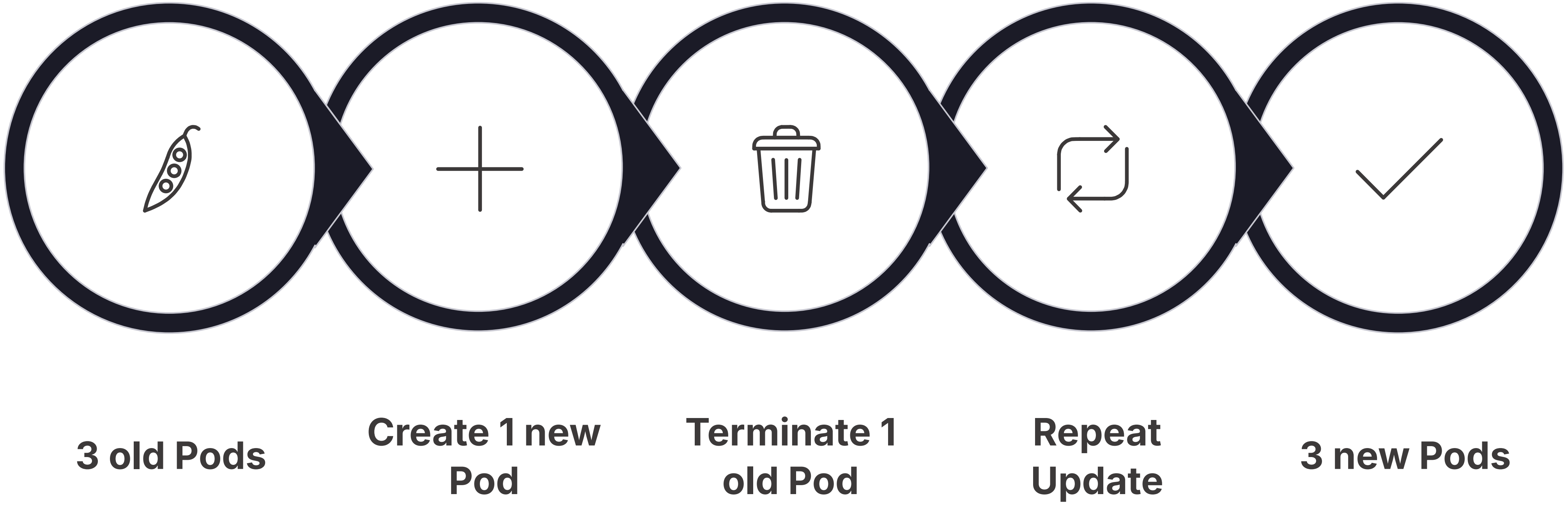
Desired State Reconciliation

Kubernetes constantly monitors actual state versus desired state. If they diverge (e.g., a Pod crashes), Kubernetes takes action to restore the desired state automatically.

MODULE 4

Rolling Updates and Rollbacks

Kubernetes enables zero-downtime deployments through rolling updates, gradually replacing old Pods with new ones. If issues arise, rollbacks restore the previous version quickly and safely.



MODULE 4

Docker vs Kubernetes

Docker and Kubernetes are complementary technologies, not competitors. Docker provides containerisation—packaging applications with dependencies. Kubernetes provides orchestration—managing those containers at scale. Understanding their relationship clarifies their respective roles.

Aspect	Docker	Kubernetes
Primary Purpose	Container runtime and image management	Container orchestration and cluster management
Scope	Single host	Multi-host cluster
Scaling	Manual, host-limited	Automatic, cluster-wide
High Availability	Not built-in	Built-in with replica sets and self-healing
Load Balancing	Requires external tools	Built-in service load balancing
Health Monitoring	Basic container health	Sophisticated probes and health checks
Updates	Stop and start containers manually	Rolling updates with zero downtime
Use Case	Development, simple deployments	Production, complex microservices
Complexity	Simple to learn and use	Steeper learning curve

ASSESSMENT

Kubernetes Knowledge Check

Evaluate your understanding of Kubernetes architecture, Pods, Services, and orchestration concepts.

Question 1

What is the smallest deployable unit in Kubernetes?

- A) Container
- B) Pod ✓
- C) Node
- D) Deployment

Explanation: A Pod is the smallest deployable unit in Kubernetes. Pods contain one or more containers and represent a single instance of a running process in the cluster.

Question 2

Which Kubernetes component is responsible for scheduling Pods to nodes?

- A) Kubelet
- B) Controller Manager
- C) Scheduler ✓
- D) API Server

Explanation: The Scheduler is responsible for assigning Pods to worker nodes based on resource requirements, constraints, and available capacity on each node.

HANDS-ON EXERCISE

Exercise 3: Kubernetes Concepts Lab

Scenario: Your organisation is deploying a web application that needs high availability and the ability to handle variable traffic. You'll use Kubernetes concepts to design a resilient deployment strategy.

Define Pod Architecture

Task: Design a Pod structure for a web application with a main container and logging sidecar.

- Main container: web server (nginx) serving the application
- Sidecar container: log collector forwarding logs to centralised system
- Shared volume: logs directory mounted in both containers

Discussion: Why use a multi-container Pod? Consider shared resources, lifecycle management, and coupling between containers.

Plan Scaling Strategy

Task: Determine appropriate scaling parameters for the deployment.

- Minimum replicas: 3 (baseline availability)
- Maximum replicas: 10 (cost constraint)
- Scaling metric: CPU utilisation threshold at 70%
- Scale-up trigger: sustained high CPU for 2 minutes
- Scale-down trigger: low CPU for 5 minutes

Discussion: Why different timeframes for scaling up vs. down? Consider traffic patterns and user experience.

Identify Failure Scenarios

Task: List failure scenarios and how Kubernetes self-healing addresses them.

Failure	Kubernetes Response
Container crash	Restart container via liveness probe
Pod becomes unresponsive	Kill and recreate Pod
Node fails	Reschedule Pods to healthy nodes
Deployment fails	Pause rollout, maintain old version

Discussion: How does self-healing improve reliability compared to manual intervention?

MODULE 5

What is Data Testing?

Data testing validates the accuracy, completeness, and quality of data as it moves through systems. Unlike functional testing which focuses on application behaviour, data testing ensures the data itself is correct, consistent, and fit for purpose. As organisations become increasingly data-driven, robust data testing practices are critical for maintaining trust in analytics, reporting, and decision-making.

Data testing encompasses validation at every stage of the data lifecycle: from initial data entry and storage, through transformations and migrations, to final consumption in reports and dashboards. It verifies that data meets business rules, maintains referential integrity, and arrives at its destination complete and unchanged (unless intentional transformations occur).

Data Accuracy

Verify that data values are correct, precise, and represent real-world entities accurately. Validate against source systems and business rules.

Data Completeness

Ensure all required data is present with no missing values, records, or fields. Validate row counts and mandatory field population.

Data Consistency

Confirm data maintains consistency across systems, with matching values in related tables and adherence to referential integrity constraints.

MODULE 5

Why Data Testing is Critical

Poor data quality has severe consequences for organisations, from incorrect business decisions to regulatory compliance failures and lost customer trust. Data testing is not optional—it's a business imperative.

Impact of Bad Data

£8.5M

Annual Cost

Average cost of poor data quality per organisation

30%

Revenue Loss

Businesses lose up to 30% revenue due to data quality issues

40%

Failed Initiatives

Of data initiatives fail due to inadequate data quality

Critical Testing Areas

- **Financial Accuracy:** Incorrect financial data leads to wrong decisions, regulatory penalties, and investor distrust
- **Customer Data:** Poor customer data quality damages relationships, reduces marketing effectiveness, and causes compliance issues
- **Operational Data:** Inaccurate operational data results in supply chain disruptions, inventory problems, and inefficiency
- **Regulatory Compliance:** Data errors can lead to compliance violations, fines, and legal consequences (GDPR, SOX, HIPAA)
- **Analytics and AI:** Machine learning models trained on bad data produce unreliable predictions and insights
- **Integration Points:** Data exchange between systems must be validated to prevent cascading errors across the organisation

MODULE 5

Types of Data Testing

Data testing encompasses multiple validation types, each addressing different aspects of data quality. Comprehensive data testing strategies incorporate all these types.

Source to Target Validation

Compares data between source and target systems to ensure accurate migration or replication. Validates row counts, column values, data types, and relationships match between systems.

Data Completeness

Verifies all expected data is present with no missing records or fields. Includes null checks, mandatory field validation, and record count reconciliation between stages.

Data Accuracy

Validates data values are correct according to business rules and constraints. Includes range checks, format validation, referential integrity, and comparison against authoritative sources.

Data Transformation Validation

Confirms transformations applied to data during processing produce expected results. Validates calculations, aggregations, data type conversions, and business logic implementations.

MODULE 5

Data Testing Concepts Explained

Data Integrity

Data integrity ensures accuracy and consistency of data over its lifecycle. It encompasses entity integrity (unique primary keys), referential integrity (valid foreign keys), domain integrity (valid values), and user-defined integrity (business rules).

- Primary keys must be unique and not null
- Foreign keys must reference existing primary keys
- Data must conform to defined constraints and rules
- Relationships between tables must be maintained

Data Reconciliation

Reconciliation compares datasets to identify discrepancies. Common scenarios include source vs. target comparison, pre-migration vs. post-migration validation, and daily reconciliation reports.

```
-- Record count comparison
SELECT COUNT(*) FROM source_table;
SELECT COUNT(*) FROM target_table;

-- Sum validation
SELECT SUM(amount) FROM source_table;
SELECT SUM(amount) FROM target_table;
```

Data Migration Testing

Migration testing validates data movement from legacy systems to new systems. It involves pre-migration analysis (data profiling, volume estimation), migration execution validation (completeness, accuracy, transformation correctness), and post-migration verification (business functionality, reporting accuracy, performance benchmarks). Critical for system upgrades, mergers and acquisitions, and cloud migrations.

MODULE 5

Real-World Banking Data Testing Example

Scenario: A retail bank is migrating customer account data from a legacy mainframe system to a new cloud-based banking platform. The migration involves 5 million customer accounts with transaction history spanning 10 years.

01

Pre-Migration Validation

- Profile source data: identify 2.3% null values in address fields, 150 duplicate accounts
- Establish baseline metrics: total accounts (5,000,000), total balance (£45.2B), transaction count (250M)
- Define acceptance criteria: 100% account migration, zero balance discrepancies, 99.99% transaction completeness

02

Migration Execution Testing

- Validate record counts: ensure all 5M accounts migrated
- Verify balance totals: confirm £45.2B total matches source
- Check account numbers: ensure no duplicates or format errors
- Validate relationships: verify transactions link correctly to accounts

03

Data Integrity Validation

- Referential integrity: confirm all transactions reference valid accounts
- Business rules: verify overdraft limits, interest rates, account status codes
- Calculated fields: validate current balance = opening balance + credits - debits
- Compliance: ensure regulatory reporting fields populated correctly

04

Post-Migration Verification

- Sample 1,000 random accounts: deep comparison between source and target
- Run production reports: compare output with legacy system reports
- User acceptance testing: bank staff validate their customer accounts
- Performance testing: ensure queries complete within SLA

ASSESSMENT

Data Testing Knowledge Check

Assess your understanding of data testing fundamentals and validation types.

Question 1

Which type of data testing would you use to verify that all records from a source system were successfully copied to a target system?

- A) Data accuracy testing
- B) Data completeness testing ✓
- C) Data transformation testing
- D) Data integrity testing

Explanation: Data completeness testing verifies all expected data is present, including validating record counts match between source and target systems.

Question 2

What does referential integrity ensure in database testing?

- A) All fields contain non-null values
- B) Foreign keys reference existing primary keys ✓
- C) Data transformations are applied correctly
- D) Record counts match between tables

Explanation: Referential integrity ensures foreign key values in one table correspond to existing primary key values in the referenced table, maintaining valid relationships between tables.

MODULE 6

What is Big Data?

Big Data refers to datasets so large or complex that traditional data processing applications cannot handle them adequately. Big Data isn't just about size—it's characterised by unique challenges in storage, processing, analysis, and visualisation requiring specialised technologies and approaches.

The term emerged in the early 2000s as organisations began collecting unprecedented volumes of data from web interactions, sensors, mobile devices, and social media. Traditional relational databases couldn't scale to handle these workloads economically, driving the development of new distributed computing frameworks like Hadoop and NoSQL databases.

> Big Data represents a paradigm shift from "sample and analyse" to "collect everything and explore". The question changes from "what should we measure?" to "what can we learn from all this data?"

MODULE 6

The 5 V's of Big Data

Big Data is characterised by five dimensions—the 5 V's—that distinguish it from traditional data processing. Understanding these characteristics is essential for designing effective Big Data testing strategies.

Volume

The sheer amount of data generated and stored. Big Data involves terabytes to petabytes of data from various sources including transactions, sensors, social media, and logs.

- Traditional DB: gigabytes to terabytes
- Big Data: terabytes to petabytes and beyond

Velocity

The speed at which data is generated and processed. Real-time or near-real-time processing is often required for streaming data from IoT devices, financial markets, and social media.

- Batch processing: hours to days
- Stream processing: seconds to milliseconds

Variety

The different types and formats of data. Big Data includes structured data (databases), semi-structured data (XML, JSON), and unstructured data (text, images, video).

- Structured: relational databases
- Unstructured: social media, documents, multimedia

Veracity

The trustworthiness and quality of data. Big Data often comes from diverse sources with varying levels of reliability, requiring validation and cleansing processes.

- Data quality issues: inconsistency, incompleteness
- Requires robust validation and cleansing

Value

The business value extracted from data. The ultimate goal is transforming raw data into actionable insights that drive business decisions and competitive advantage.

- Raw data has potential value
- Analysis and insights create actual business value

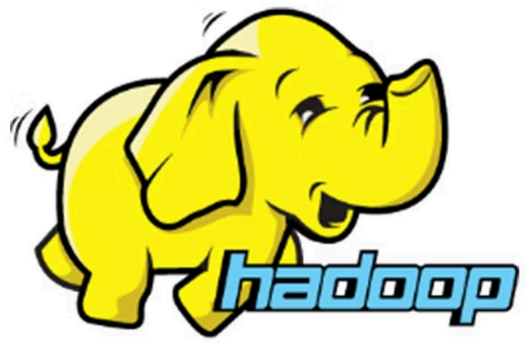
MODULE 6

What is Hadoop?

Apache Hadoop is an open-source framework for distributed storage and processing of massive datasets across clusters of commodity computers. Developed by Yahoo in the mid-2000s and inspired by Google's MapReduce and Google File System papers, Hadoop revolutionised Big Data processing by making it economical and accessible.

Hadoop enables organisations to store and process petabytes of data using clusters of inexpensive servers rather than expensive supercomputers. It achieves this through distributed computing—dividing large datasets into smaller chunks, distributing them across many nodes, processing in parallel, and combining results.

The framework automatically handles hardware failures, data replication, and job scheduling, allowing developers to focus on business logic rather than infrastructure management. This fault tolerance and scalability made Hadoop the foundation of Big Data ecosystems.



Hadoop's Impact: Before Hadoop, processing terabytes of data required expensive proprietary systems. Hadoop democratised Big Data, enabling startups and enterprises alike to analyse massive datasets economically.

MODULE 6

Hadoop Ecosystem Overview

Hadoop isn't a single technology—it's an ecosystem of components working together to provide comprehensive Big Data capabilities. Understanding core components and their roles is essential for Big Data testing.

HDFS

Hadoop Distributed File System stores data across multiple machines with automatic replication for fault tolerance.

MapReduce

Programming model for processing large datasets in parallel across the cluster using map and reduce functions.

YARN

Yet Another Resource Negotiator manages cluster resources and job scheduling, enabling multiple processing frameworks.

Hive

Data warehouse infrastructure providing SQL-like query language (HiveQL) for analysing data stored in HDFS.

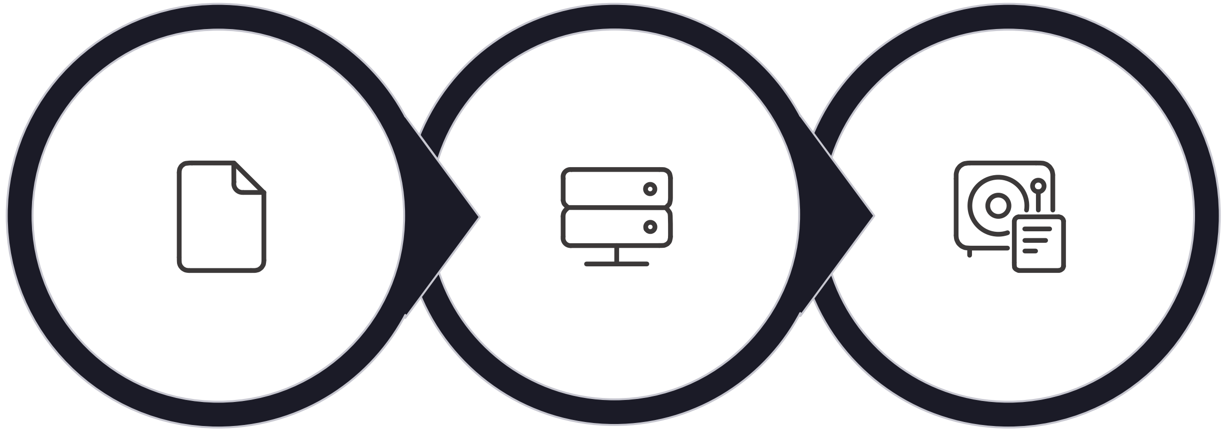
Spark

Fast, general-purpose cluster computing system providing in-memory processing for faster analytics than MapReduce.

MODULE 6

HDFS: Hadoop Distributed File System

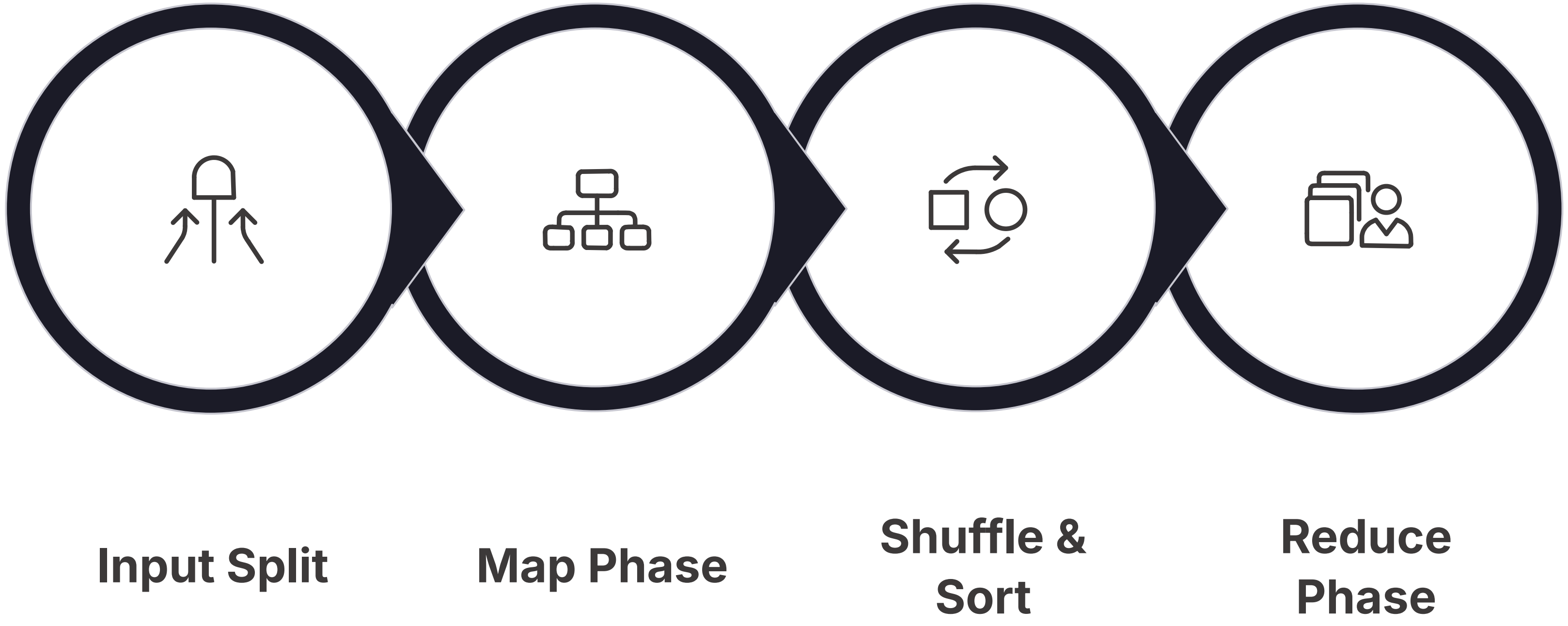
HDFS is designed to store very large files across multiple machines whilst providing high throughput access and fault tolerance. Understanding HDFS architecture is crucial for testing Big Data applications.



MODULE 6

MapReduce Conceptual Overview

MapReduce is a programming model for processing large datasets in parallel. It consists of two main phases: Map (filter and sort data) and Reduce (aggregate results). Whilst largely replaced by Spark for many use cases, understanding MapReduce concepts remains important.



MODULE 6

YARN: Resource Management

YARN (Yet Another Resource Negotiator) is Hadoop's cluster resource management system. It separates resource management from data processing, allowing multiple processing frameworks to share the same Hadoop cluster efficiently.

YARN Components

- **ResourceManager:** Global resource scheduler allocating resources across all applications running on the cluster
- **NodeManager:** Per-node agent monitoring resource usage and reporting to ResourceManager
- **ApplicationMaster:** Per-application coordinator negotiating resources and working with NodeManagers to execute tasks
- **Containers:** Allocation of resources (CPU, memory, disk) on a single node for executing application tasks

YARN enables running multiple processing frameworks (MapReduce, Spark, Tez) on the same cluster, improving resource utilisation and reducing operational overhead.

Resource Allocation Flow

01

Client submits application to ResourceManager

02

ResourceManager allocates container for ApplicationMaster

03

ApplicationMaster negotiates resources from ResourceManager

04

ApplicationMaster instructs NodeManagers to launch containers

05

Application executes tasks in containers

MODULE 6

Traditional Database vs Hadoop

Traditional relational databases and Hadoop serve different purposes and excel in different scenarios. Understanding when to use each is crucial for architectural decisions and testing strategies.

Characteristic	Traditional RDBMS	Hadoop
Data Structure	Structured, schema-on-write	Any format, schema-on-read
Data Size	Gigabytes to terabytes	Terabytes to petabytes
Processing Model	Interactive queries, transactions	Batch processing, analytics
Access Speed	Low latency, seconds	High throughput, minutes to hours
Scaling	Vertical (bigger hardware)	Horizontal (more nodes)
Data Updates	Frequent updates and deletes	Write once, read many
Best For	Transactional workloads, OLTP	Analytics, data lakes, ETL processing
Cost	Expensive proprietary licences	Commodity hardware, open source
Data Integrity	ACID transactions	Eventual consistency

ASSESSMENT

Big Data & Hadoop Knowledge Check

Evaluate your understanding of Big Data concepts and Hadoop ecosystem components.

Question 1

Which of the 5 V's of Big Data refers to the trustworthiness and quality of data?

- A) Volume
- B) Velocity
- C) Variety
- D) Veracity ✓

Explanation: Veracity refers to the trustworthiness, accuracy, and quality of data. Big Data often comes from diverse sources with varying reliability, requiring validation processes.

Question 2

What is the primary purpose of HDFS in the Hadoop ecosystem?

- A) Execute MapReduce jobs
- B) Manage cluster resources
- C) Distributed storage with fault tolerance ✓
- D) Provide SQL query interface

Explanation: HDFS (Hadoop Distributed File System) provides distributed storage for large files across multiple machines with automatic replication for fault tolerance.

MODULE 7

What is ETL?

ETL stands for Extract, Transform, Load—a data integration process that moves data from source systems into a data warehouse or data lake. ETL is fundamental to business intelligence, analytics, and data-driven decision-making, consolidating data from disparate sources into a unified, consistent format for analysis.

The ETL process has existed for decades but remains critical in modern data architectures. Whether loading traditional data warehouses, cloud data lakes, or real-time analytics platforms, understanding ETL principles is essential for data testing professionals.

Extract

Pull data from source systems including databases, APIs, files, and applications

Transform

Cleanse, standardise, enrich, and restructure data to meet business requirements

Load

Insert transformed data into target data warehouse or data lake for analysis

MODULE 7

ETL Process Deep Dive

Each ETL phase involves specific activities and challenges. Understanding what happens at each stage is crucial for comprehensive testing strategies.

Extract Phase

Purpose: Retrieve data from source systems whilst minimising impact on operational systems.

- **Full Extraction:** Complete data pull from source, typically for initial loads
- **Incremental Extraction:** Only changed or new data since last extract, reducing load
- **Change Data Capture (CDC):** Track changes in real-time using database logs or triggers

Challenges: Source system performance impact, handling different data formats, managing extraction windows, dealing with source system downtime

Transform Phase

Purpose: Convert extracted data into format suitable for analysis and business intelligence.

- **Cleansing:** Remove duplicates, fix errors, handle missing values
- **Standardisation:** Ensure consistent formats (dates, addresses, phone numbers)
- **Enrichment:** Add derived columns, lookup values, calculated fields
- **Aggregation:** Summarise detailed data to appropriate granularity
- **Business Rules:** Apply validation and transformation logic

Challenges: Complex business logic, performance with large volumes, maintaining data quality, handling exceptions

Load Phase

Purpose: Insert transformed data into target warehouse efficiently.

- **Full Load:** Truncate and reload entire table, simple but disruptive
- **Incremental Load:** Insert only new/changed records, maintaining history
- **Upsert:** Update existing records or insert if not present
- **Slowly Changing Dimensions:** Track historical changes in dimension tables

Challenges: Load window constraints, maintaining referential integrity, handling load failures, optimising bulk load performance

MODULE 7

ETL Architecture Components

A typical ETL architecture includes several key components working together to move and transform data reliably.

Architecture Layers

- **Source Systems:** Operational databases, legacy systems, external data feeds, flat files, APIs
- **Staging Area:** Temporary storage for extracted data, isolation from source and target
- **ETL Engine:** Transformation logic, business rules, data quality checks
- **Data Warehouse:** Target repository optimised for analytics with star/snowflake schemas
- **Metadata Repository:** Stores ETL job definitions, lineage, execution history
- **Monitoring & Logging:** Tracks job execution, errors, data quality metrics

Data Warehouse Structure

- **Fact Tables:** Store quantitative transaction data (sales, orders, events)
- **Dimension Tables:** Store descriptive attributes (customers, products, time, geography)
- **Data Marts:** Subject-specific subsets for departmental analysis
- **Aggregate Tables:** Pre-calculated summaries for performance

Staging Area Purpose: Provides isolation for transformation logic, enables recovery from failures, and reduces load on source and target systems.

MODULE 7

ETL Testing: Validation Types

Comprehensive ETL testing validates data at every stage—from source extraction through transformation logic to final warehouse loading. Different validation types address different quality aspects.

Data Completeness Validation

Verify all expected records are extracted, transformed, and loaded. Compare record counts between source and target, validate no data loss during processing, and ensure all mandatory fields are populated.

```
-- Record count validation
SELECT COUNT(*) FROM source_table; -- 1,000,000
SELECT COUNT(*) FROM target_table; -- 1,000,000
```

Transformation Logic Validation

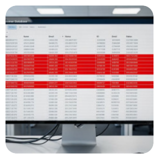
Confirm business rules and calculations are applied correctly. Test data type conversions, aggregations, derived columns, lookup enrichments, and complex business logic implementations.

```
-- Validate calculation
SELECT SUM(unit_price * quantity) FROM source;
SELECT SUM(total_amount) FROM target;
```

Null Value Validation

Identify unexpected null values in mandatory fields. Verify null handling logic, default value assignments, and ensure critical fields required for analysis are not null.

```
-- Check for nulls in critical fields
SELECT COUNT(*) FROM target_table
WHERE customer_id IS NULL OR order_date IS NULL;
```



Duplicate Record Checks

Ensure no duplicate records exist in the target. Validate primary key uniqueness, check for duplicate business keys, and verify de-duplication logic works correctly.

```
-- Find duplicates
SELECT customer_id, COUNT(*)
FROM target_table
GROUP BY customer_id
HAVING COUNT(*) > 1;
```

Performance Validation

Verify ETL jobs complete within acceptable timeframes. Monitor execution times, identify bottlenecks, validate incremental loads are faster than full loads, and ensure processes meet SLAs.