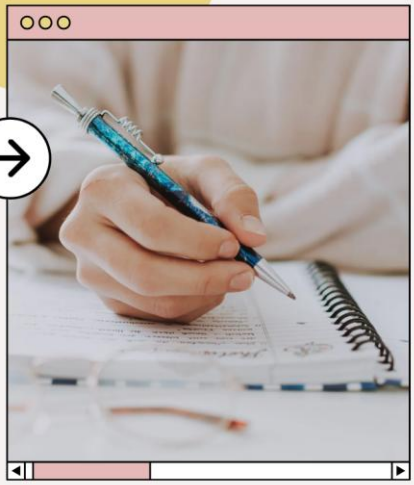
UNIT 1 - INTRODUCTION TO DEVOPS & VERSION CONTROL - FN



Devops

A culture and set of practices that combine software development (Dev) and IT operations (Ops).

↘ OVERVIEW

A culture and set of practices that combine development (Dev) and IT operations (Ops)

↘ WHY

- Traditional challenges: slow releases, “throw over the wall” mindset, poor collaboration.

DevOps breaks silos between Dev, QA, and Ops.

↘ CHARACTERISTICS

- Collaboration across teams
- Automation at every stage
- Continuous delivery of value
- Faster feedback loops

↘ BENEFITS

- Faster time to market
- Higher software quality
- Improved reliability & uptime
- Better alignment with business goals

DevOps Lifecycle

Phases (infinity loop model):

1. Plan – Define features, backlog (Jira, Azure Boards)
2. Code – Write code, manage versions (Git, GitHub)
3. Build – Compile, package applications (Maven, Gradle, Jenkins)
4. Test – Automated testing for quality (JUnit, Selenium)
5. Release – Versioning and preparing artifacts (GitHub Actions, Jenkins)
6. Deploy – Push to production/staging (Docker, Kubernetes, Ansible)
7. Operate – Ensure uptime, scaling (Kubernetes, AWS, Azure)
8. Monitor – Collect feedback and logs (Prometheus, Grafana, ELK stack)

Agile & DevOps

↘ AGILE

Iterative development, sprints, customer collaboration

↘ AGILE & DEVOPS COMPLEMENT

- Agile: focuses on what to build and how to build it.
- DevOps: focuses on how to deliver & operate software continuously.

↘ KEY DIFFERENCE

Agile ≠ DevOps → Agile is about methodology, DevOps is about culture + tools.

↘ INDUSTRY EXAMPLE

- Agile sprint produces features → DevOps pipeline deploys features automatically

Git Basics

↘ WHY

- Tracks code changes
- Enables collaboration
- Prevents accidental loss of work

↘ GIT CONCEPTS

- Repository: storage location for project (local & remote)
- Working Directory: actual files on disk
- Staging Area (Index): temporary area for preparing commits
- Commit: snapshot of changes in history

↘ GIT WORKFLOW

Modify files → Stage (git add) → Commit (git commit) → Push to remote (git push)

↘ BASIC GIT COMMANDS

- git init – Initialize repo
- git clone <url> – Copy repo from remote
- git add <file> – Stage changes
- git commit -m "message" – Commit changes
- git push / git pull – Sync with remote repo

Remote Repositor-ies

↘ CONCEPT

Why teams need a central remote repository (GitHub/GitLab)

↘ COLLABORATION WORKFLOW

Local → Remote → Team members

↘ KEY OPERATIONS

push, pull, fetch

↘ PRACTICAL USAGE

open-source contributions, enterprise projects

Branching Strategies

↘ WHY BRANCHES

Parallel feature development, experimentation, isolation

↘ COMMON STRATEGIES

- Feature Branching – each feature on its own branch.
- Git Flow – long-lived develop branch, release/hotfix branches.
- Trunk-Based Development – commits directly to main with feature toggles.

↘ BEST PRACTICES

- Keep branches small & short-lived.
- Regularly sync with main branch.
- Use meaningful branch names (feature/login-ui, bugfix/cart-crash).

↘ PRACTICAL USAGE

Frequent Releases, Long Release Cycle

Merging & Resolving Conflicts

↘ MERGE TYPES:

- Fast-forward merge
- Three-way merge

↘ WHY CONFLICTS HAPPEN

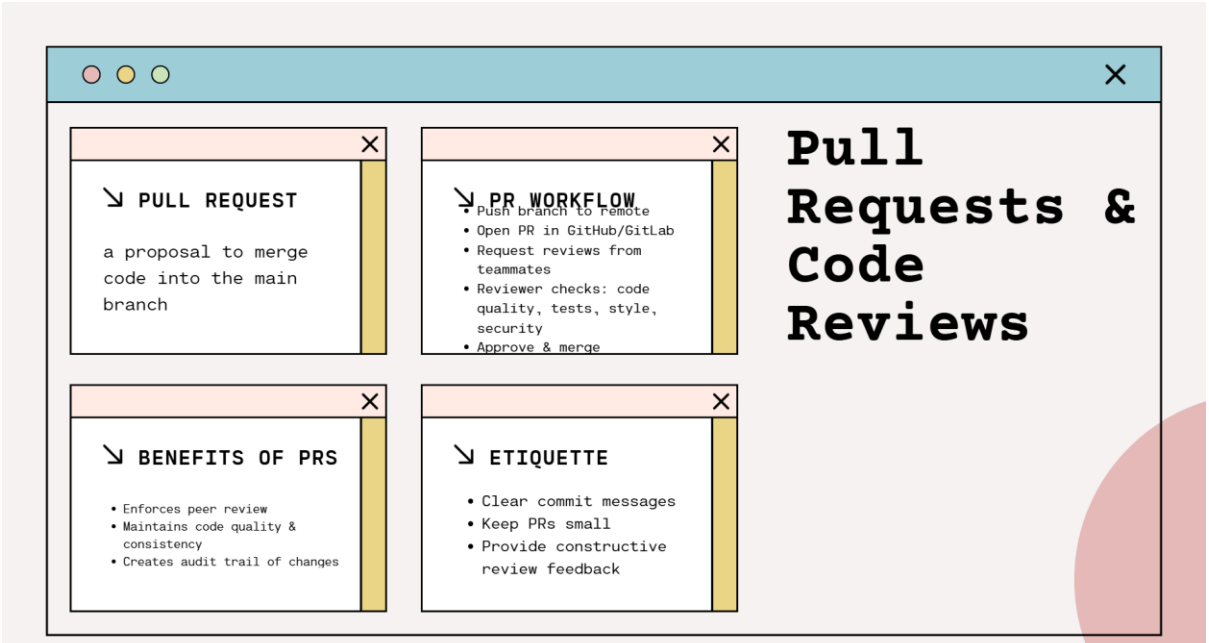
two people edit same line/file.

↘ CONFLICT RESOLUTION WORKFLOW:

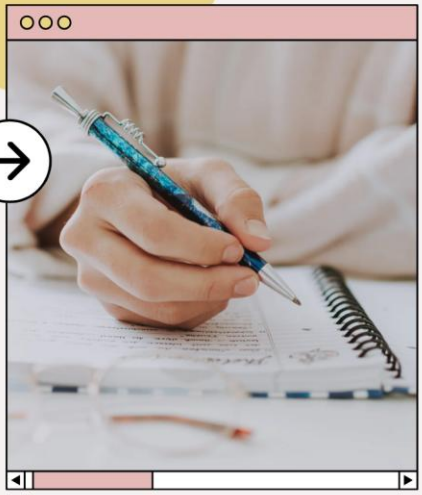
- dentify (git status, conflict markers in file)
- Edit file manually or via IDE
- Stage (git add) & commit resolved version

↘ BEST PRACTICES TO AVOID CONFLICTS

frequent pulls, smaller commits, clear ownership


↘ PULL REQUEST
a proposal to merge code into the main branch
↘ PR WORKFLOW
• Push branch to remote
• Open PR in GitHub/GitLab
• Request reviews from teammates
• Reviewer checks: code quality, tests, style, security
• Approve & merge
Pull Requests & Code Reviews
↘ BENEFITS OF PRS
• Enforces peer review
• Maintains code quality & consistency
• Creates audit trail of changes
↘ ETIQUETTE
• Clear commit messages
• Keep PRs small
• Provide constructive review feedback

UNIT 2 - CONTINUOUS INTEGRATION WITH JENKINS - FN



CI/CD Fundamentals

↘ CONTINUOUS INTEGRATION (CI)

Practice where developers frequently integrate code into a shared repository to detect issues early

↘ CONTINUOUS DEPLOYMENT

Automatically deploying to production without manual approva

↘ BENEFITS

Faster releases, fewer bugs, consistent quality, and better collaboration

↘ CI/CD PIPELINE STAGES

Build → Test → Deploy → Release (end-to-end automation cycle

Introduction to Jenkins

↘ DEFINITION

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

↘ KEY FEATURES

1,800+ plugins, pipeline-as-code, distributed builds, integrations with Git, Docker, Kubernetes

↘ USE CASES

Automating builds, running tests, deploying to servers/cloud, and monitoring pipelines

↘ ALTERNATIVES

ArgoCD, Kubevela

Jenkins Architecture

↘ MASTER/CONTROLLER

Orchestrates jobs, manages configurations, schedules tasks

↘ AGENTS/NODES

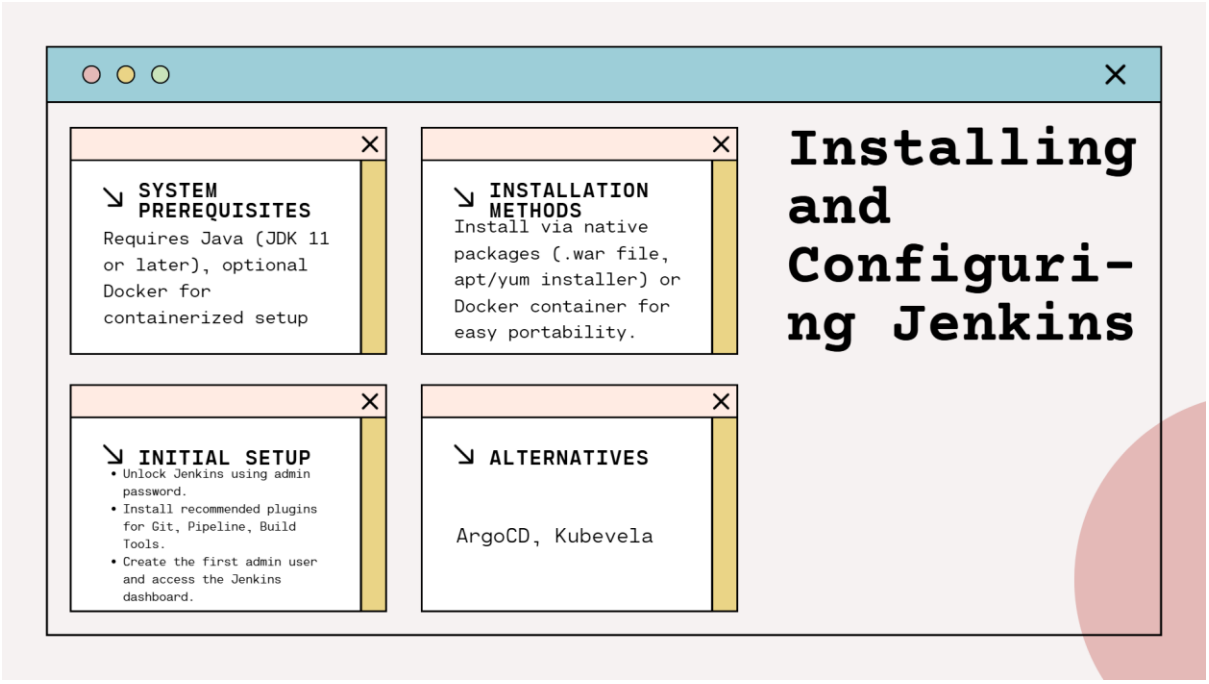
Execute jobs on separate machines (can be physical/VMs/containers)

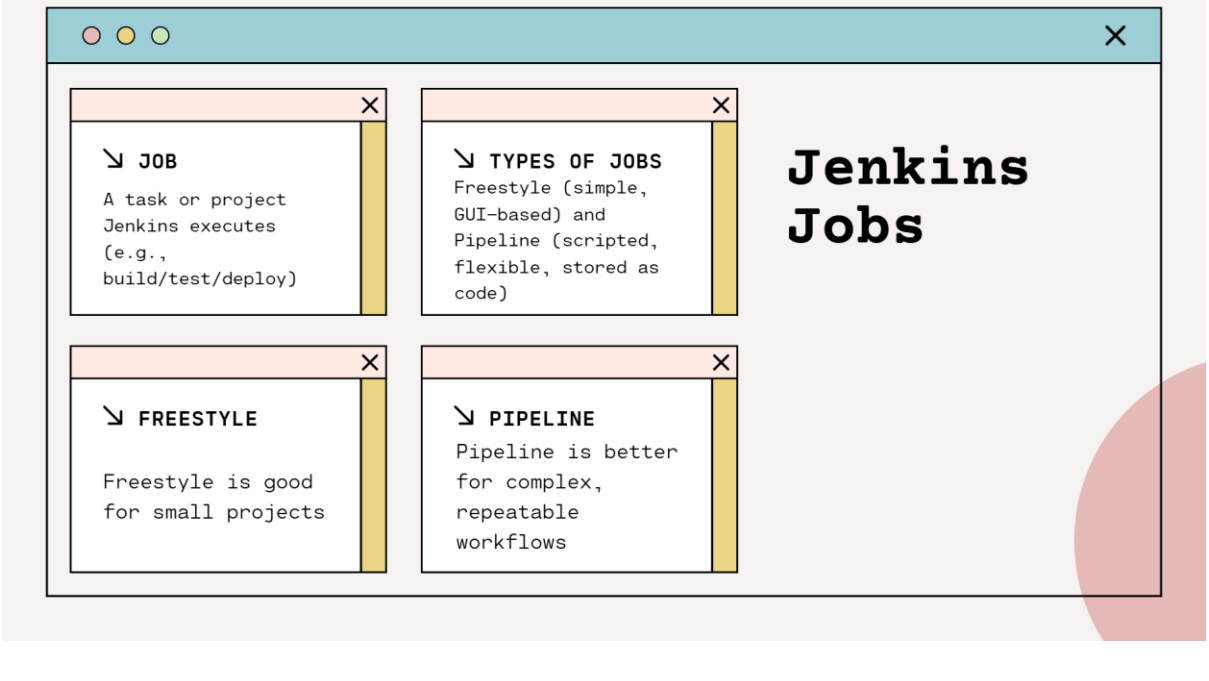
↘ DISTRIBUTED BUILDS

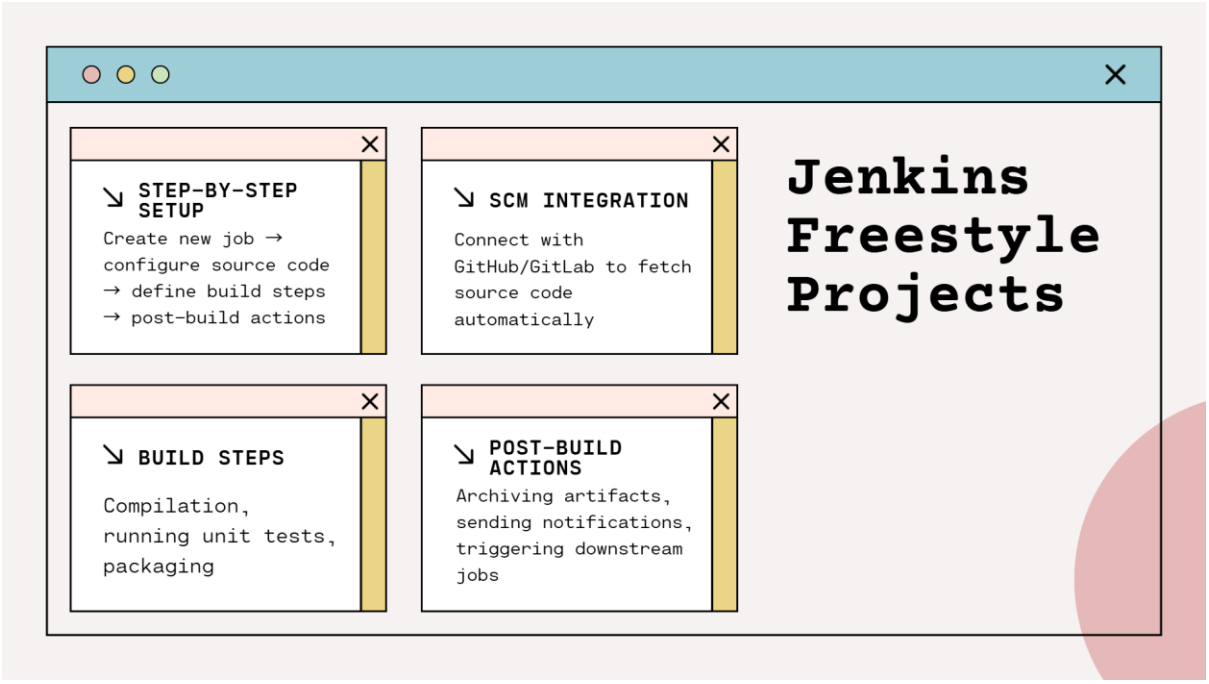
Allows parallel builds, load balancing, and platform-specific testing.

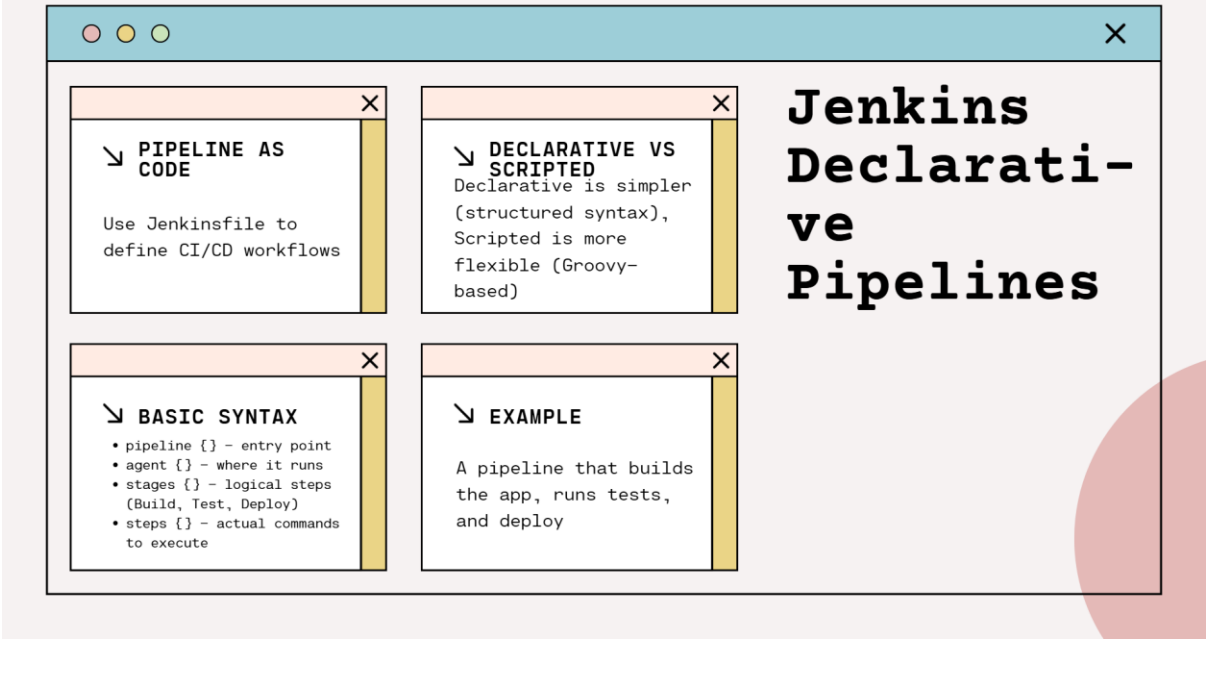
↘ EXAMPLE

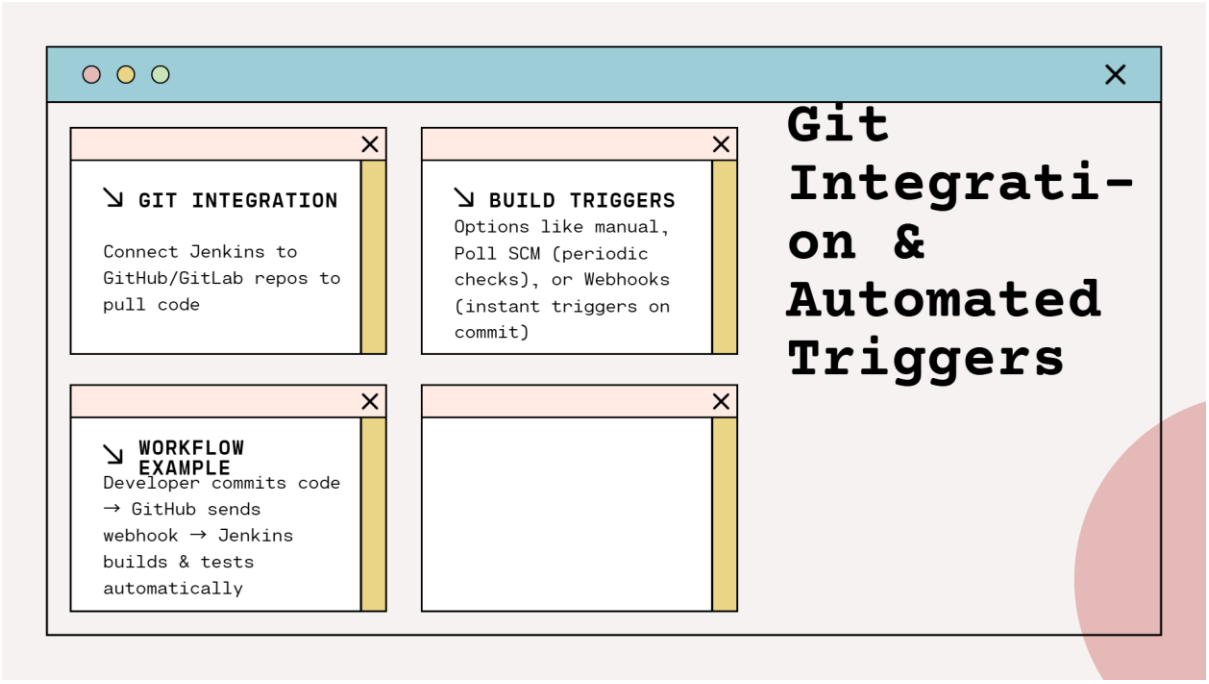
Jenkins master on one server + multiple agents for Java, Python, and Docker builds


↘ SYSTEM PREREQUISITES
Requires Java (JDK 11 or later), optional Docker for containerized setup
↘ INSTALLATION METHODS
Install via native packages (.war file, apt/yum installer) or Docker container for easy portability.
Installing and Configuri-ng Jenkins
↘ INITIAL SETUP
• Unlock Jenkins using admin password.
• Install recommended plugins for Git, Pipeline, Build Tools.
• Create the first admin user and access the Jenkins dashboard.
↘ ALTERNATIVES
ArgoCD, Kubevela

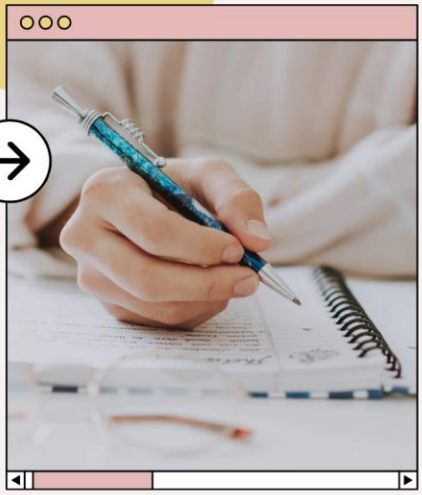

↘ JOB
A task or project Jenkins executes (e.g., build/test/deploy)
↘ TYPES OF JOBS
Freestyle (simple, GUI-based) and Pipeline (scripted, flexible, stored as code)
Jenkins Jobs
↘ FREESTYLE
Freestyle is good for small projects
↘ PIPELINE
Pipeline is better for complex, repeatable workflows


↘ STEP-BY-STEP SETUP
Create new job → configure source code → define build steps → post-build actions
↘ SCM INTEGRATION
Connect with GitHub/GitLab to fetch source code automatically
↘ BUILD STEPS
Compilation, running unit tests, packaging
↘ POST-BUILD ACTIONS
Archiving artifacts, sending notifications, triggering downstream jobs
Jenkins Freestyle Projects


↘ PIPELINE AS CODE
Use Jenkinsfile to define CI/CD workflows
↘ DECLARATIVE VS SCRIPTED
Declarative is simpler (structured syntax), Scripted is more flexible (Groovy-based)
↘ BASIC SYNTAX
• pipeline {} – entry point
• agent {} – where it runs
• stages {} – logical steps (Build, Test, Deploy)
• steps {} – actual commands to execute
↘ EXAMPLE
A pipeline that builds the app, runs tests, and deploy
Jenkins Declarati-ve Pipelines


↘ GIT INTEGRATION
Connect Jenkins to GitHub/GitLab repos to pull code
↘ BUILD TRIGGERS
Options like manual, Poll SCM (periodic checks), or Webhooks (instant triggers on commit)
↘ WORKFLOW EXAMPLE
Developer commits code → GitHub sends webhook → Jenkins builds & tests automatically
Git Integrati-on & Automated Triggers

UNIT 3 - CONTAINERIZATION WITH DOCKER - FN



Introduction to Containers

↘ DEFINITION

OS-level virtualization to run isolated applications

↘ LIGHTWEIGHT

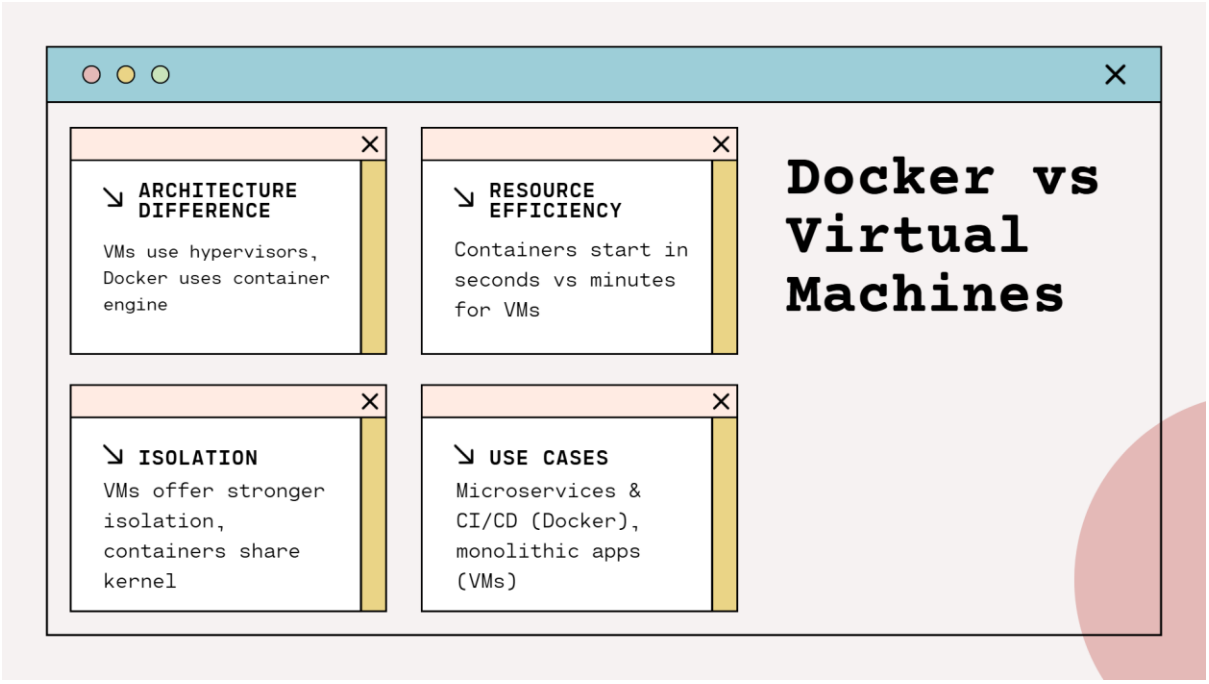
share the host OS kernel, faster than VMs

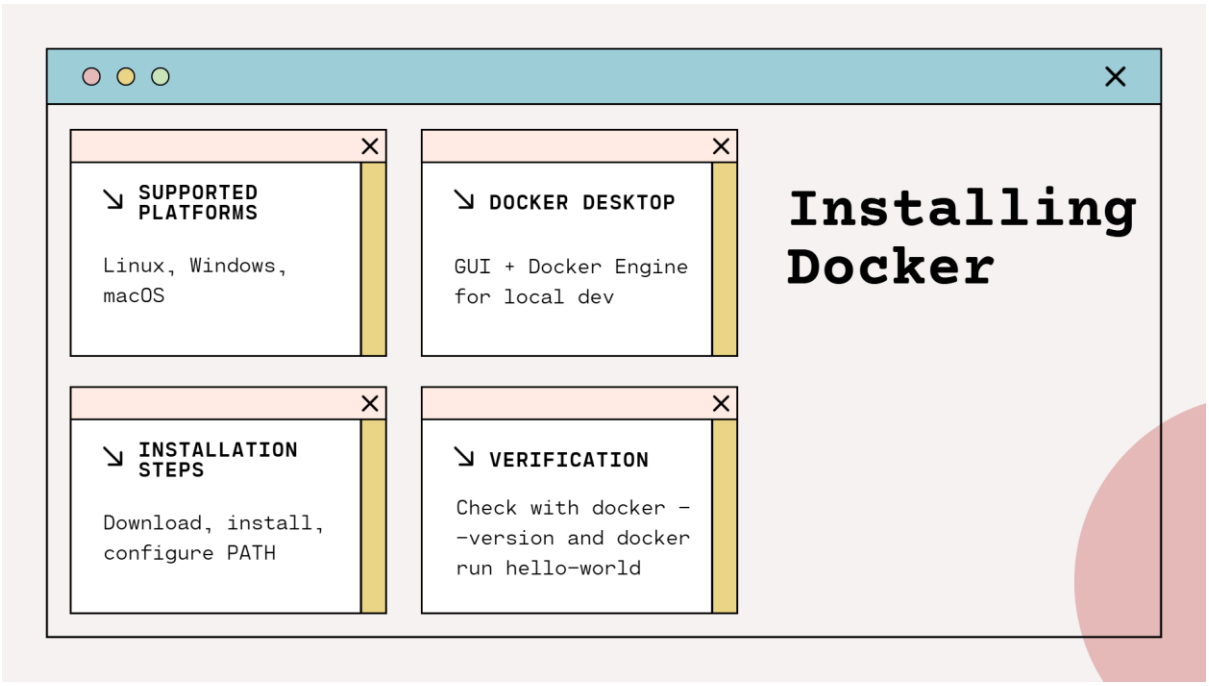
↘ PORTABILITY

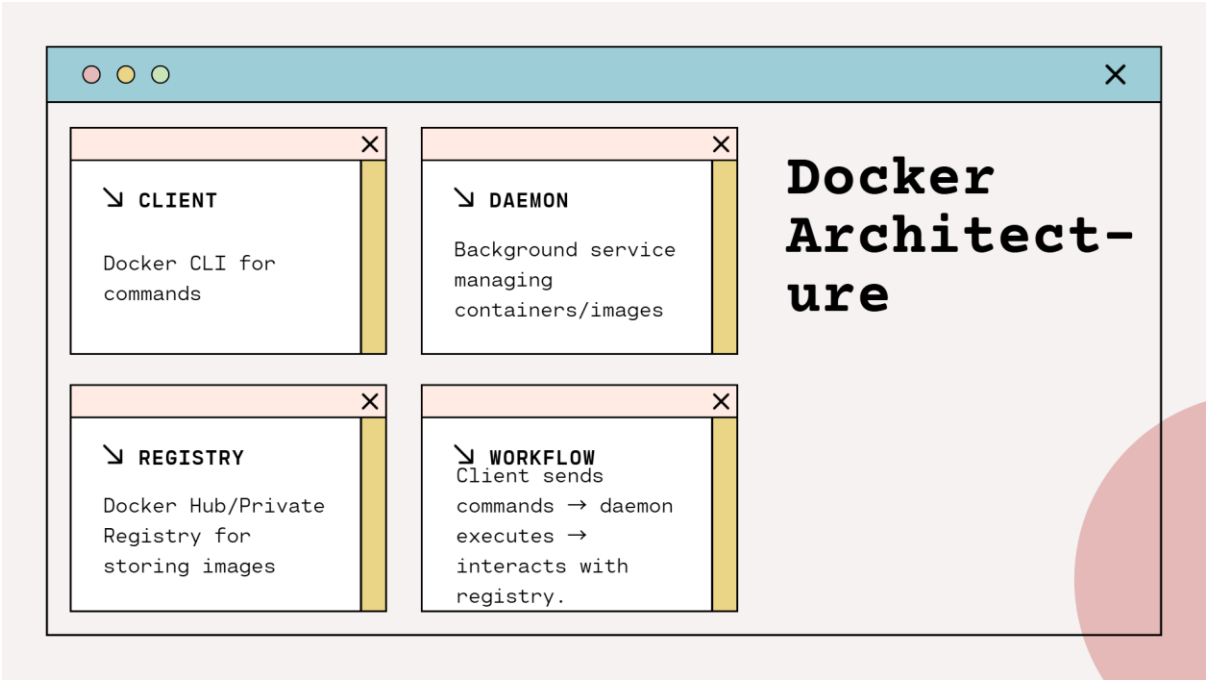
Run the same container image across environments

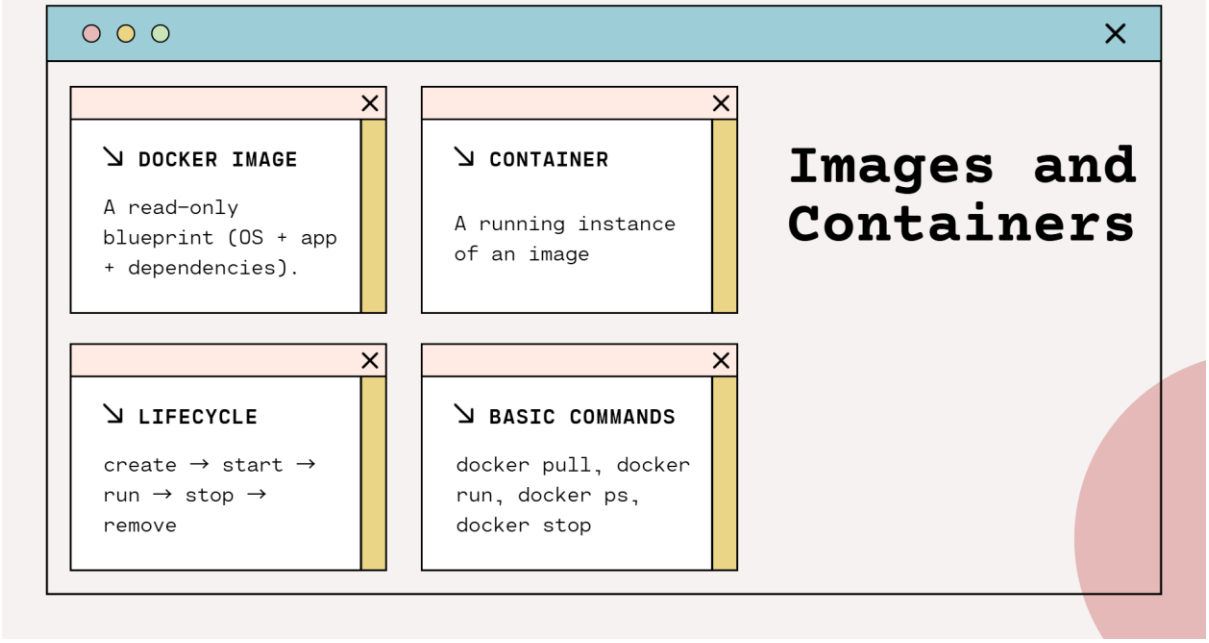
↘ PIPELINE

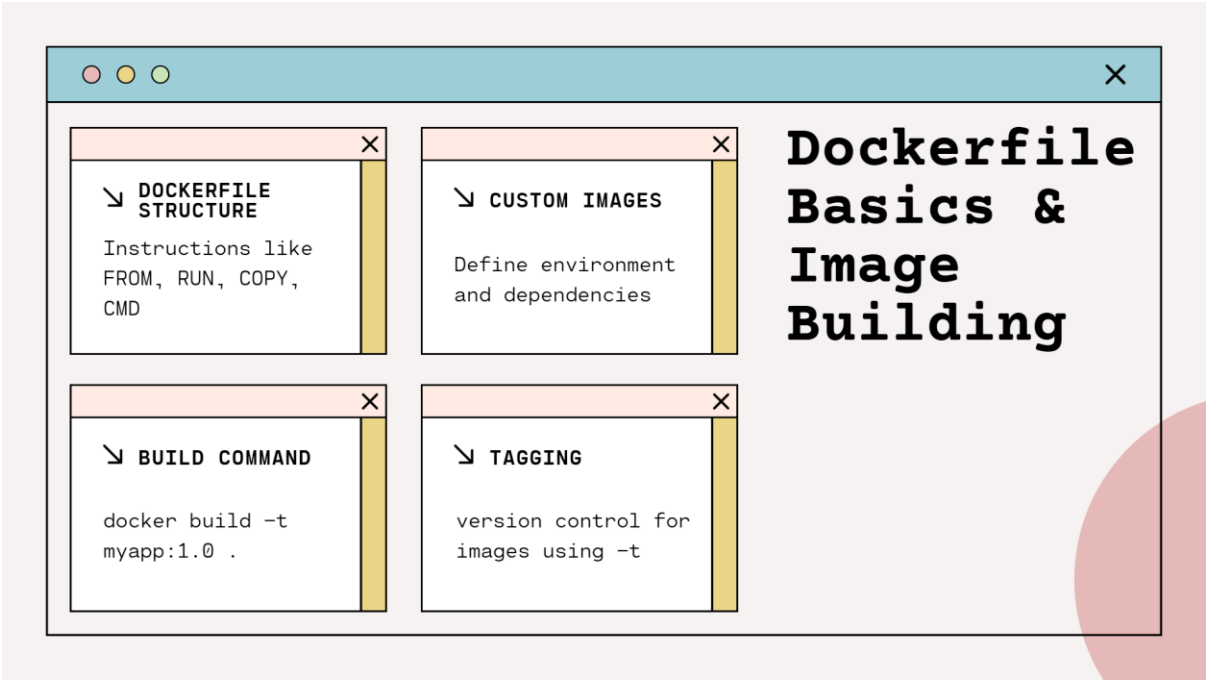
Improves consistency in build → test → deploy cycles

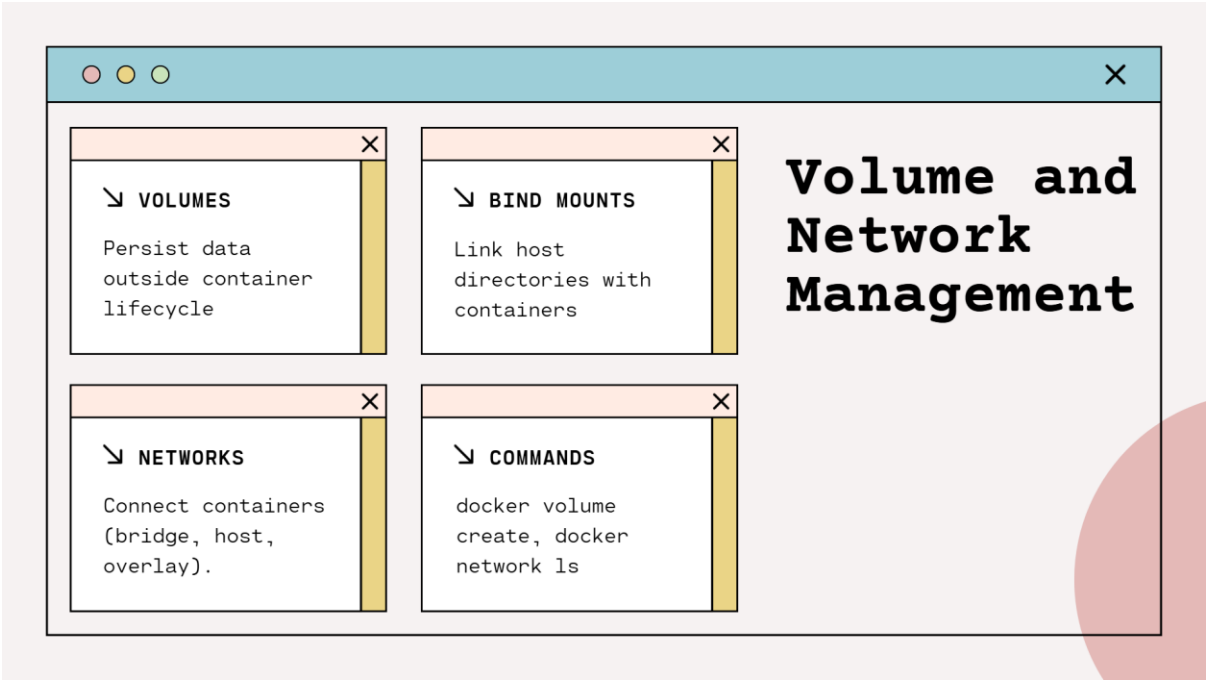

Docker vs Virtual Machines
ARCHITECTURE DIFFERENCE
VMs use hypervisors, Docker uses container engine
RESOURCE EFFICIENCY
Containers start in seconds vs minutes for VMs
ISOLATION
VMs offer stronger isolation, containers share kernel
USE CASES
Microservices & CI/CD (Docker), monolithic apps (VMs)

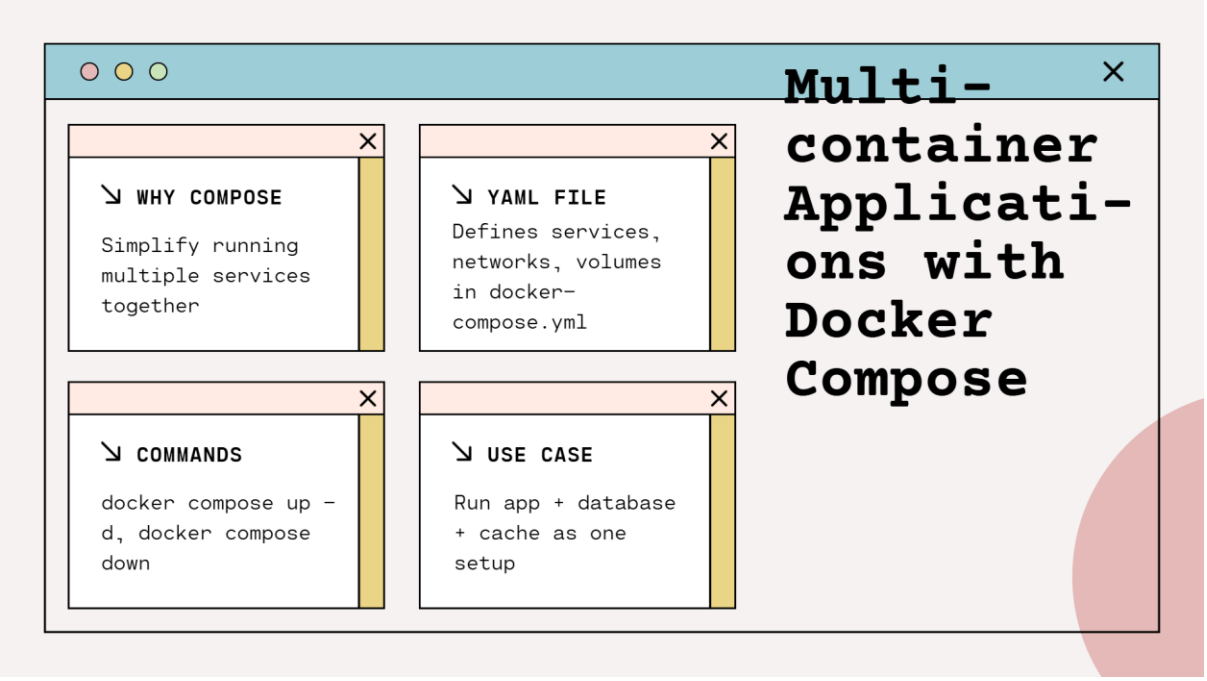

Installing Docker
SUPPORTED PLATFORMS
Linux, Windows, macOS
DOCKER DESKTOP
GUI + Docker Engine for local dev
INSTALLATION STEPS
Download, install, configure PATH
VERIFICATION
Check with docker --version and docker run hello-world


↘ CLIENT
Docker CLI for commands
↘ DAEMON
Background service managing containers/images
↘ REGISTRY
Docker Hub/Private Registry for storing images
↘ WORKFLOW
Client sends commands → daemon executes → interacts with registry.
Docker Architect-ure

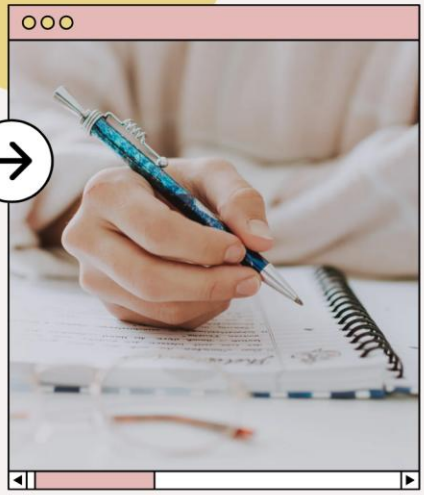

↘ DOCKER IMAGE
A read-only blueprint (OS + app + dependencies).
↘ CONTAINER
A running instance of an image
↘ LIFECYCLE
create → start → run → stop → remove
↘ BASIC COMMANDS
docker pull, docker run, docker ps, docker stop
Images and Containers


DOCKERFILE STRUCTURE
Instructions like FROM, RUN, COPY, CMD
CUSTOM IMAGES
Define environment and dependencies
BUILD COMMAND
docker build -t myapp:1.0 .
TAGGING
version control for images using -t
Dockerfile Basics & Image Building


VOLUMES
Persist data outside container lifecycle
BIND MOUNTS
Link host directories with containers
NETWORKS
Connect containers (bridge, host, overlay).
COMMANDS
docker volume create, docker network ls
Volume and Network Management


Multi-container Applicati-ons with Docker Compose
↘ WHY COMPOSE
Simplify running multiple services together
↘ YAML FILE
Defines services, networks, volumes in docker-compose.yml
↘ COMMANDS
docker compose up -d, docker compose down
↘ USE CASE
Run app + database + cache as one setup

UNIT 4 - CONTAINER ORCHESTRATION WITH KUBERNETES - FN



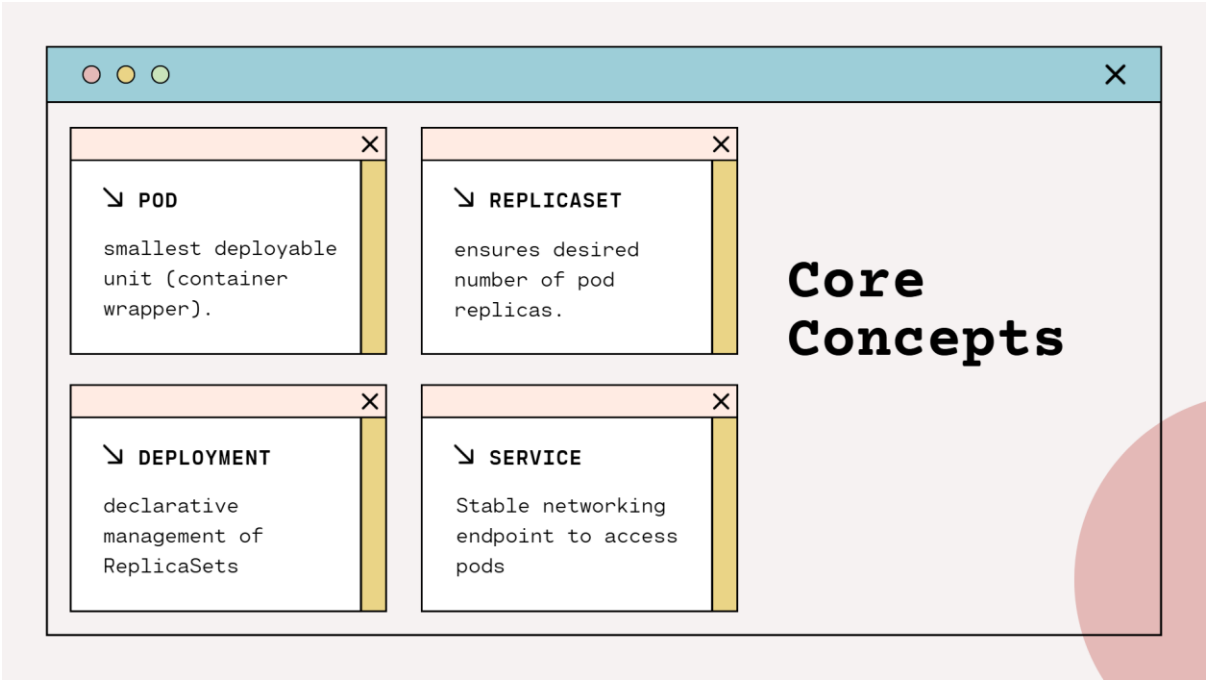
Kubernetes Overview

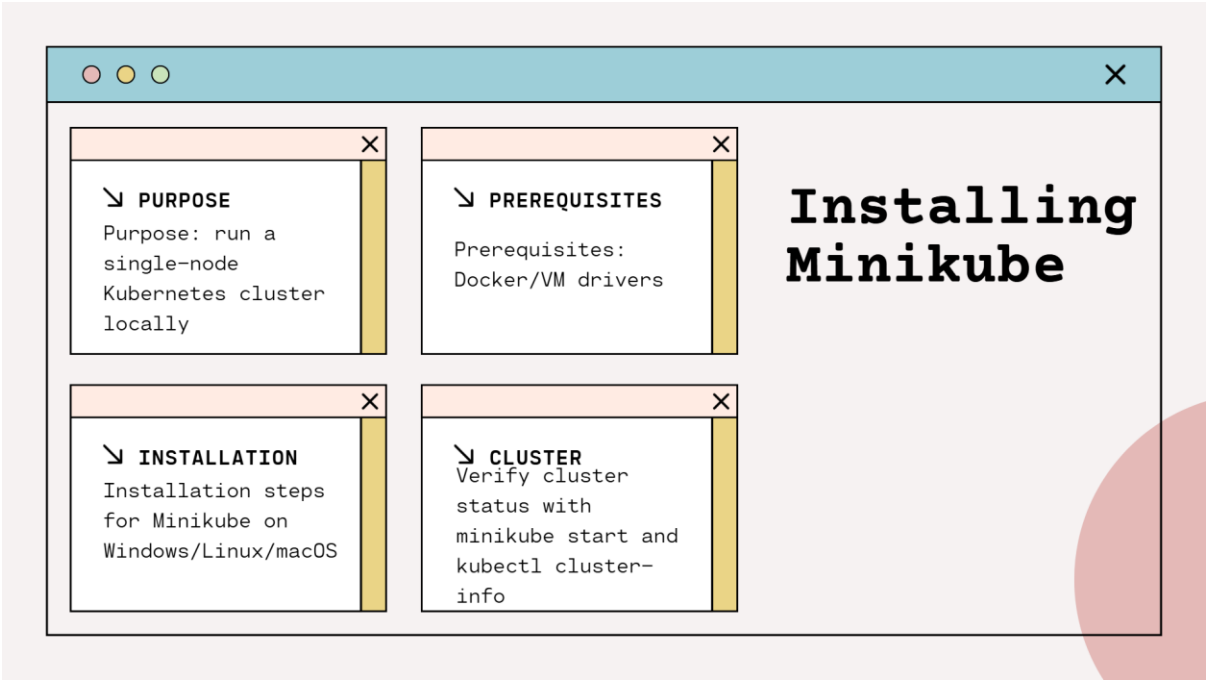
↘ INTRODUCTION
Introduction to Kubernetes as a container orchestration platform

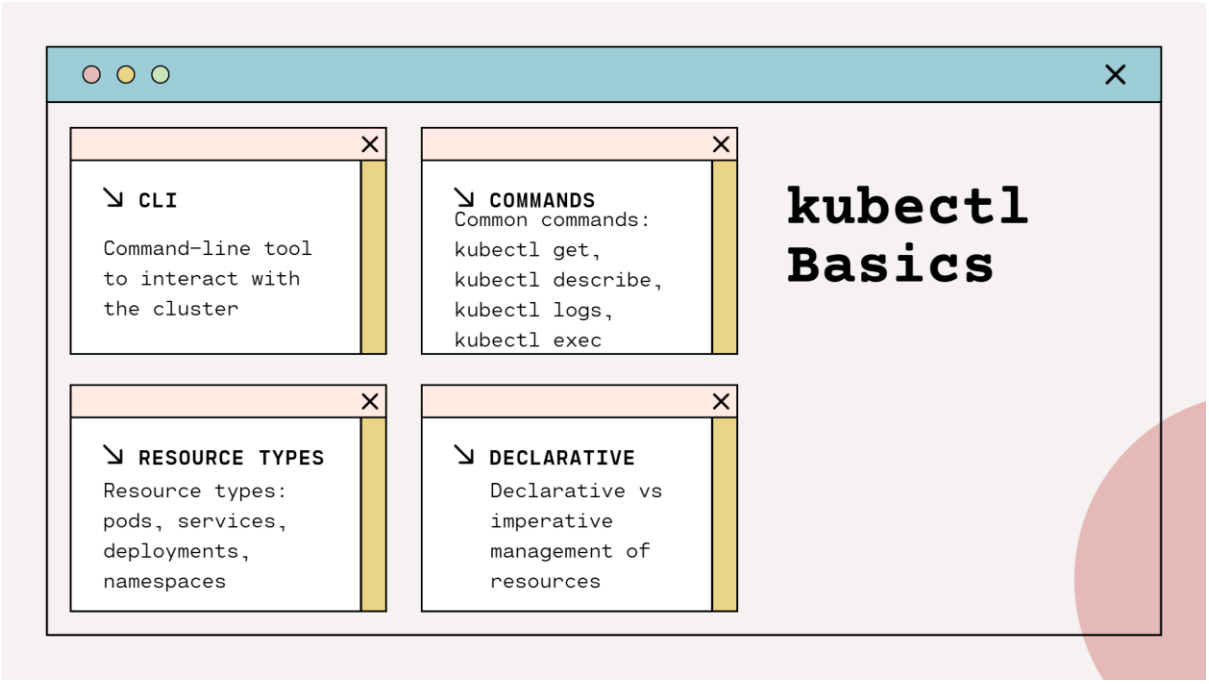
↘ BENEFITS
Benefits: scalability, self-healing, automated rollouts/rollbacks

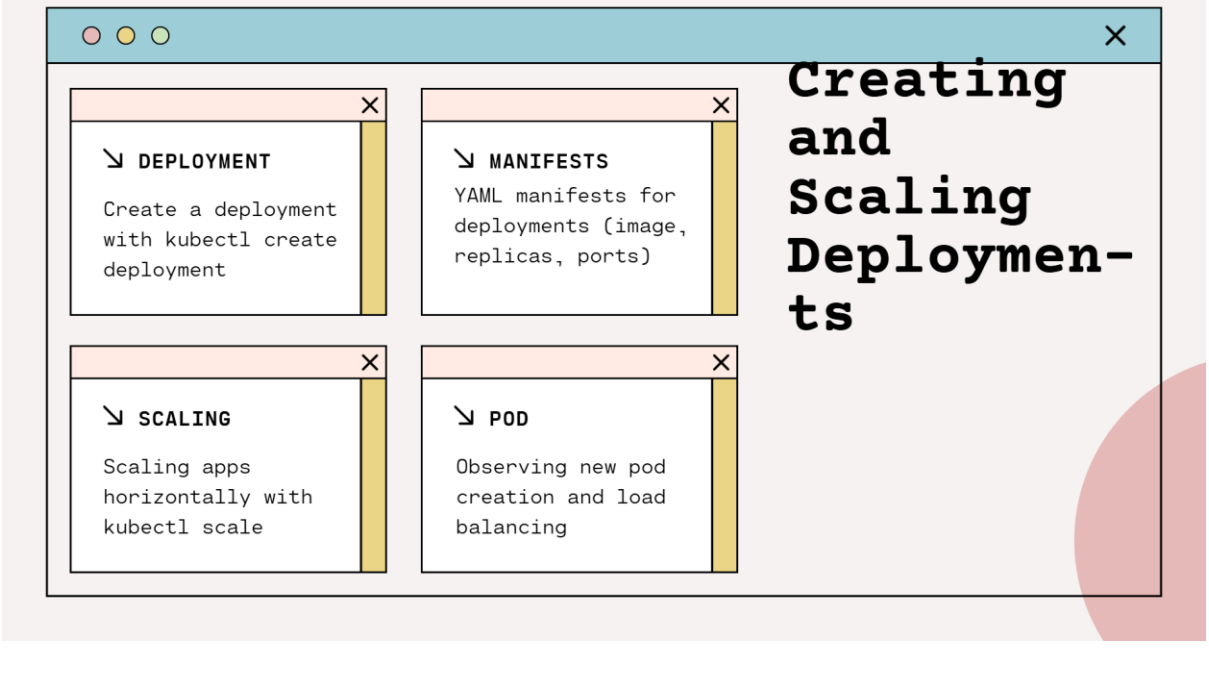
↘ COMPARISON
Comparison with Docker Compose (single host vs multi-node clusters)

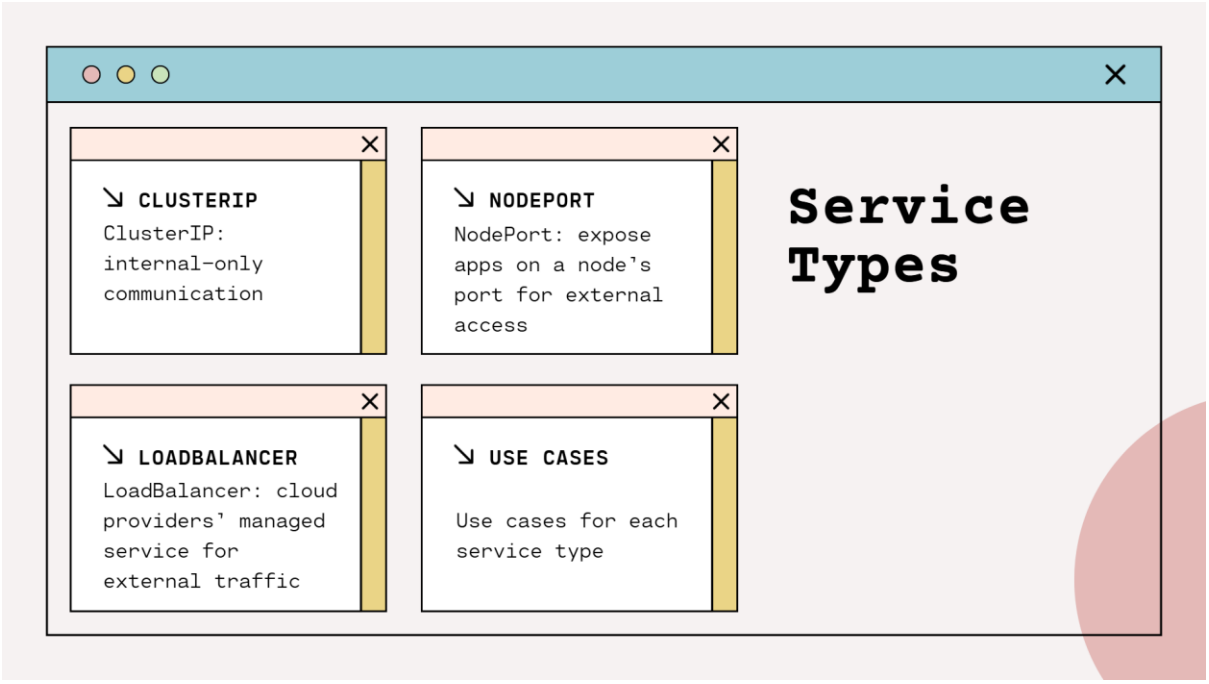
↘ USE CASES
Real-world use cases in DevOps and production

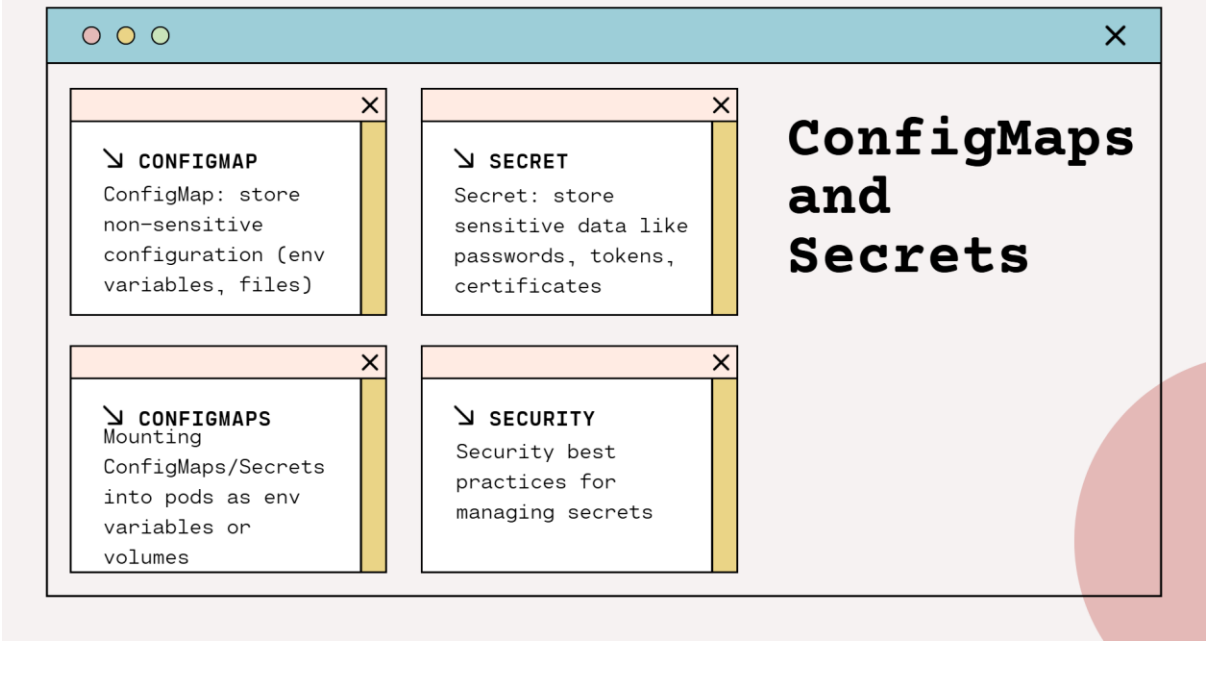

↘ POD
smallest deployable unit (container wrapper).
↘ REPLICASET
ensures desired number of pod replicas.
↘ DEPLOYMENT
declarative management of ReplicaSets
↘ SERVICE
Stable networking endpoint to access pods
Core Concepts

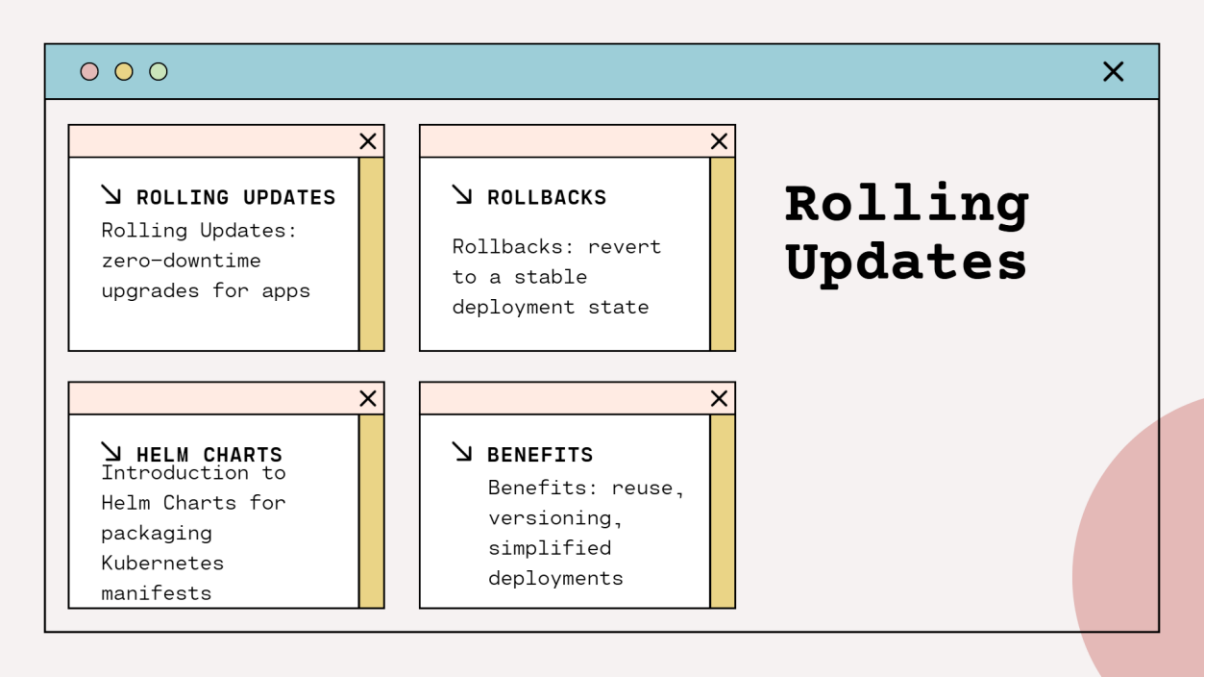

↘ PURPOSE
Purpose: run a single-node Kubernetes cluster locally
↘ PREREQUISITES
Prerequisites: Docker/VM drivers
↘ INSTALLATION
Installation steps for Minikube on Windows/Linux/macOS
↘ CLUSTER
Verify cluster status with minikube start and kubectl cluster-info
Installing Minikube


↘ CLI
Command-line tool to interact with the cluster
↘ COMMANDS
Common commands: kubectl get, kubectl describe, kubectl logs, kubectl exec
↘ RESOURCE TYPES
Resource types: pods, services, deployments, namespaces
↘ DECLARATIVE
Declarative vs imperative management of resources
kubectl Basics

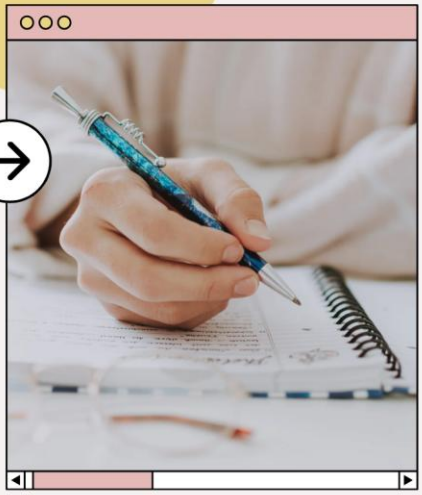

↘ DEPLOYMENT
Create a deployment with kubectl create deployment
↘ MANIFESTS
YAML manifests for deployments (image, replicas, ports)
↘ SCALING
Scaling apps horizontally with kubectl scale
↘ POD
Observing new pod creation and load balancing
Creating and Scaling Deploymen-ts


↘ CLUSTERIP
ClusterIP: internal-only communication
↘ NODEPORT
NodePort: expose apps on a node's port for external access
↘ LOADBALANCER
LoadBalancer: cloud providers' managed service for external traffic
↘ USE CASES
Use cases for each service type
Service Types


↘ CONFIGMAP
ConfigMap: store non-sensitive configuration (env variables, files)
↘ SECRET
Secret: store sensitive data like passwords, tokens, certificates
↘ CONFIGMAPS
Mounting ConfigMaps/Secrets into pods as env variables or volumes
↘ SECURITY
Security best practices for managing secrets
ConfigMaps and Secrets


↘ ROLLING UPDATES
Rolling Updates: zero-downtime upgrades for apps
↘ ROLLBACKS
Rollbacks: revert to a stable deployment state
Rolling Updates
↘ HELM CHARTS
Introduction to Helm Charts for packaging Kubernetes manifests
↘ BENEFITS
Benefits: reuse, versioning, simplified deployments

UNIT 5 -CLOUD DEPLOYMENT & MONITORING - FN



Cloud Platforms for DevOps

↘ AWS FOR DEVOPS

EC2, EKS, S3, CodePipeline, container registry (ECR)

↘ AZURE DEVOPS SERVICES

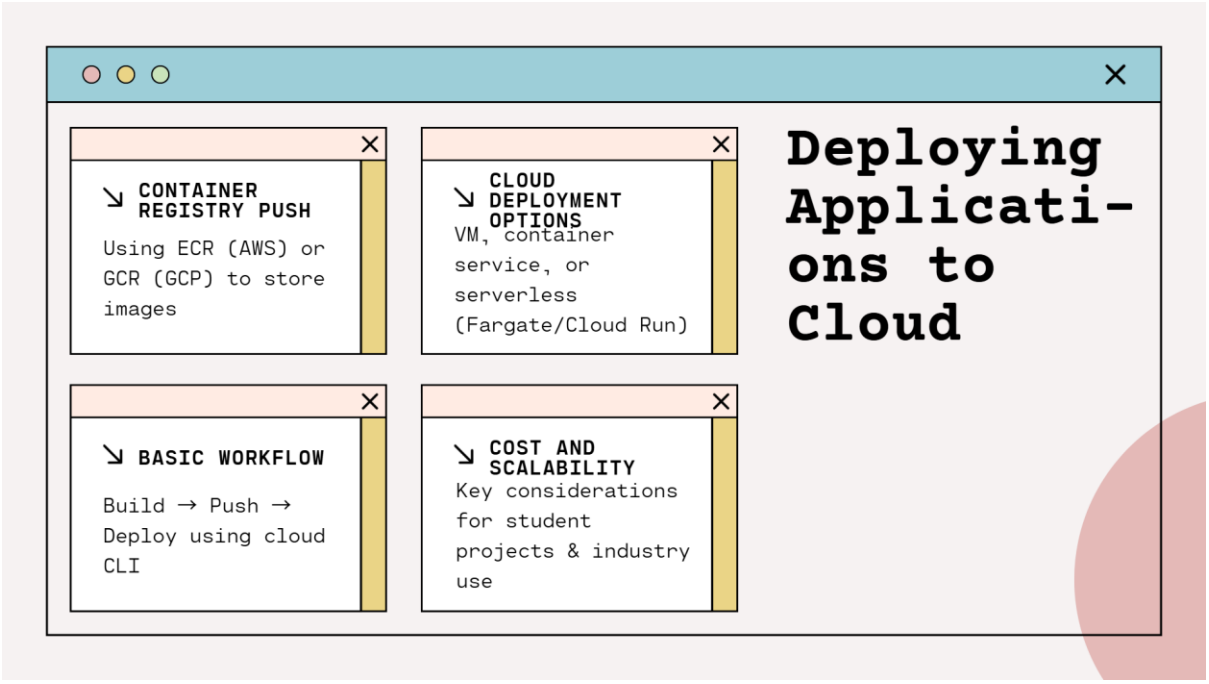
Azure Pipelines, AKS, monitoring tools

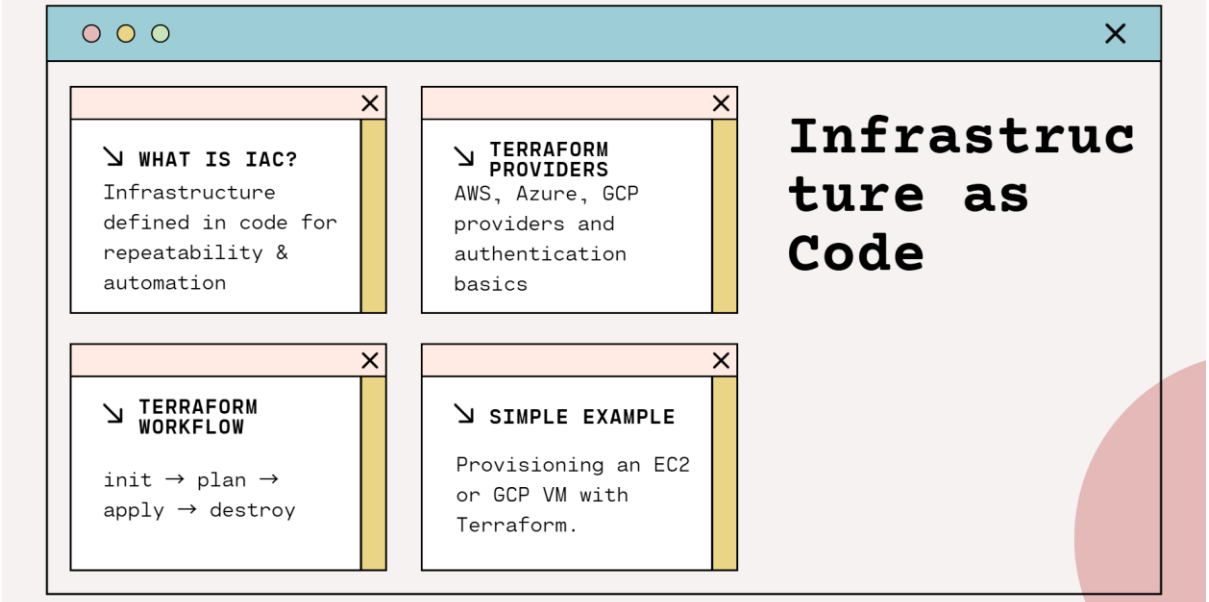
↘ GOOGLE CLOUD

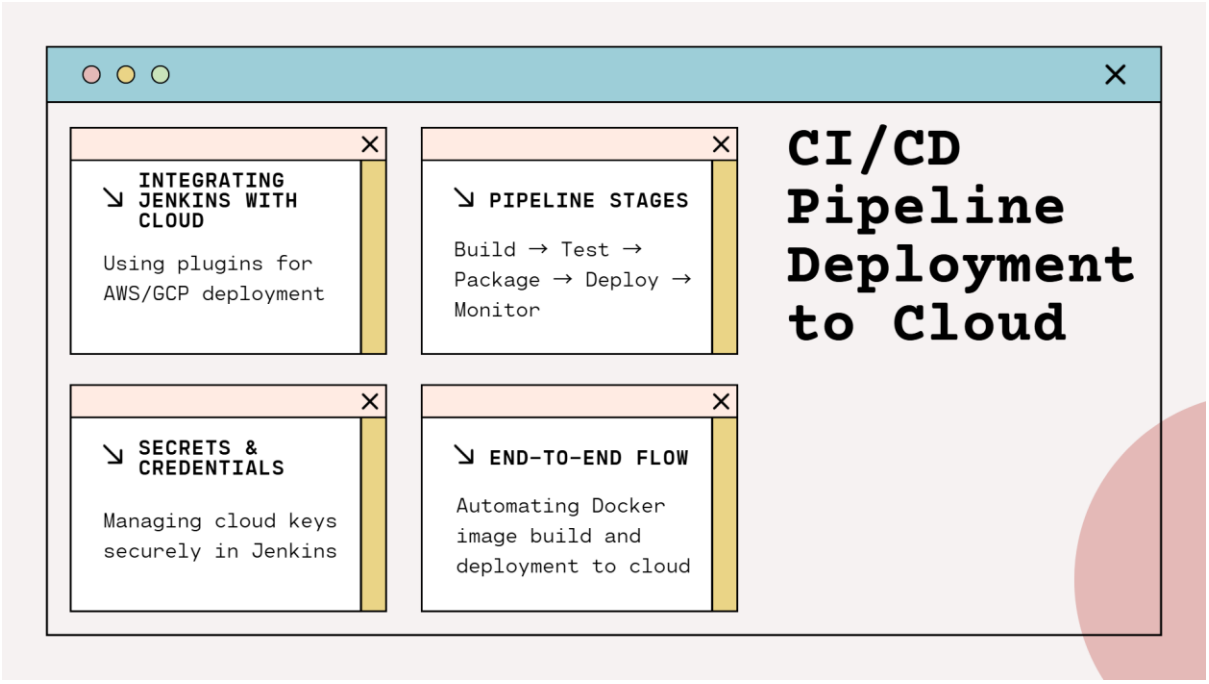
GKE, Cloud Build, Artifact Registry for DevOps

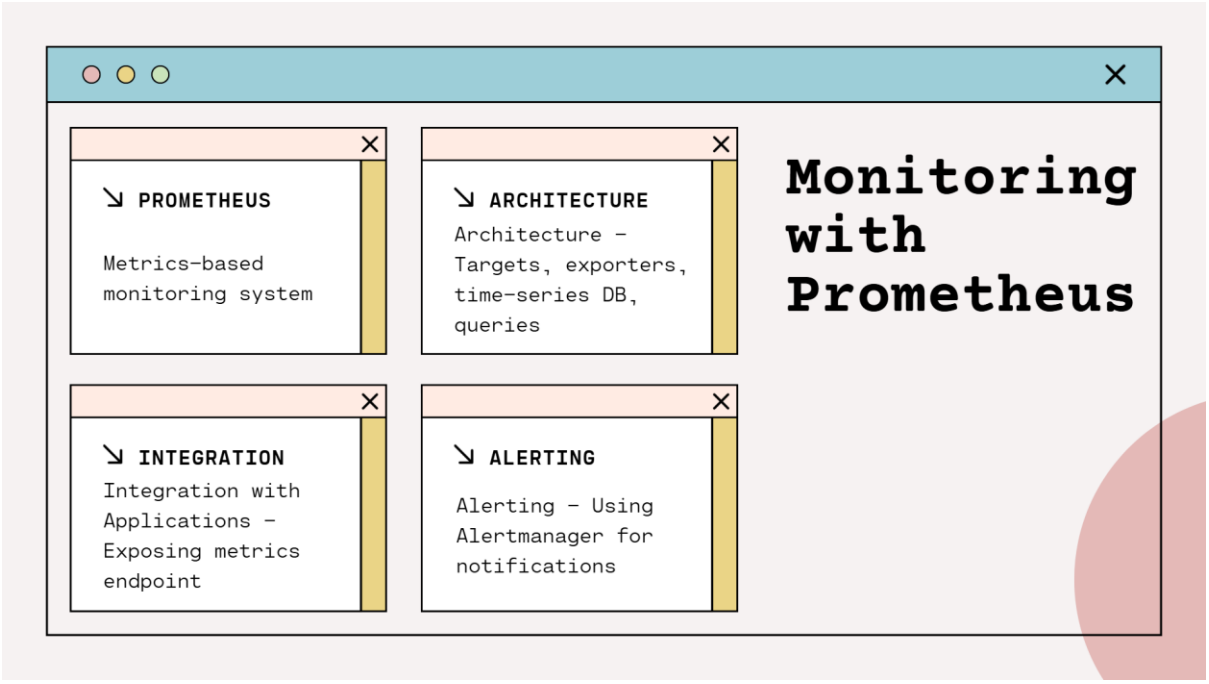
↘ COMPARISON

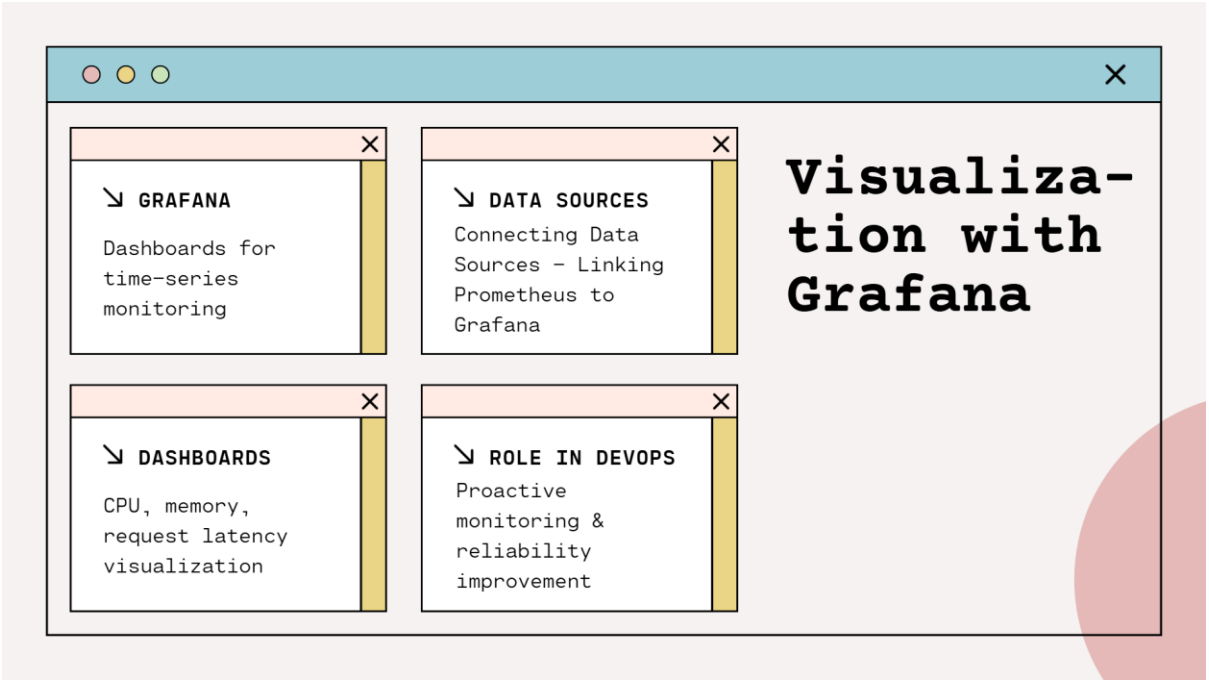
When to choose AWS vs Azure vs GCP for different workloads

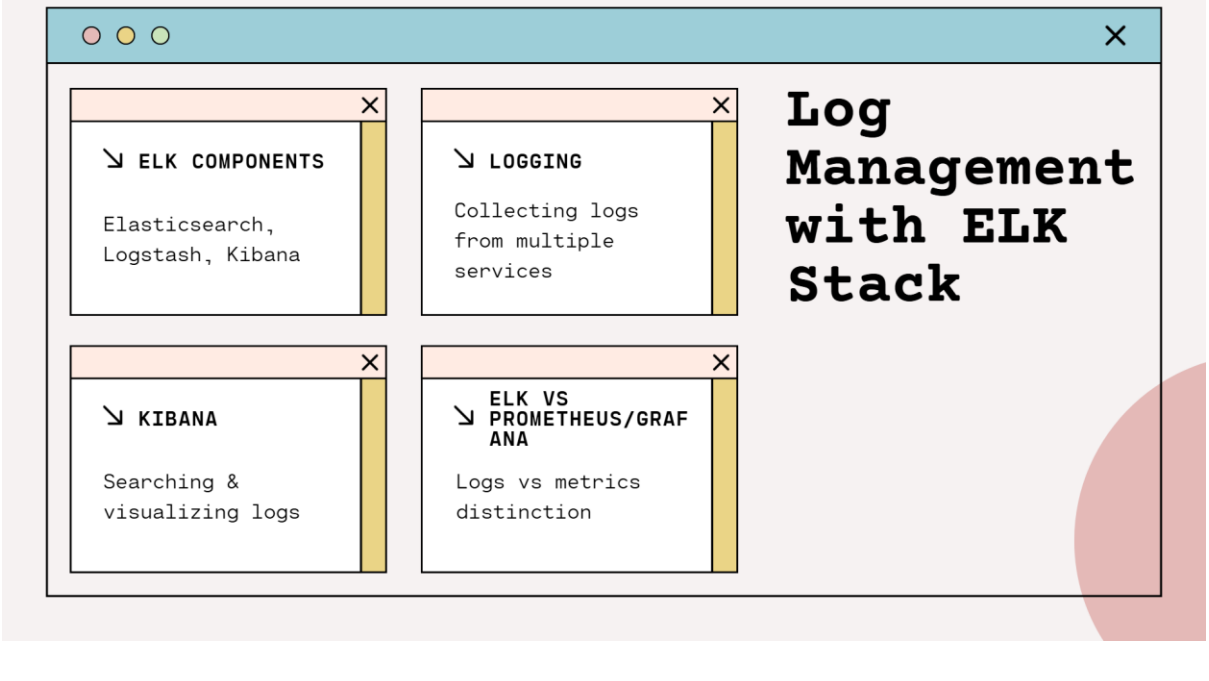

↘ CONTAINER REGISTRY PUSH
Using ECR (AWS) or GCR (GCP) to store images
↘ CLOUD DEPLOYMENT OPTIONS
VM, container service, or serverless (Fargate/Cloud Run)
↘ BASIC WORKFLOW
Build → Push → Deploy using cloud CLI
↘ COST AND SCALABILITY
Key considerations for student projects & industry use
Deploying Applicati-ons to Cloud

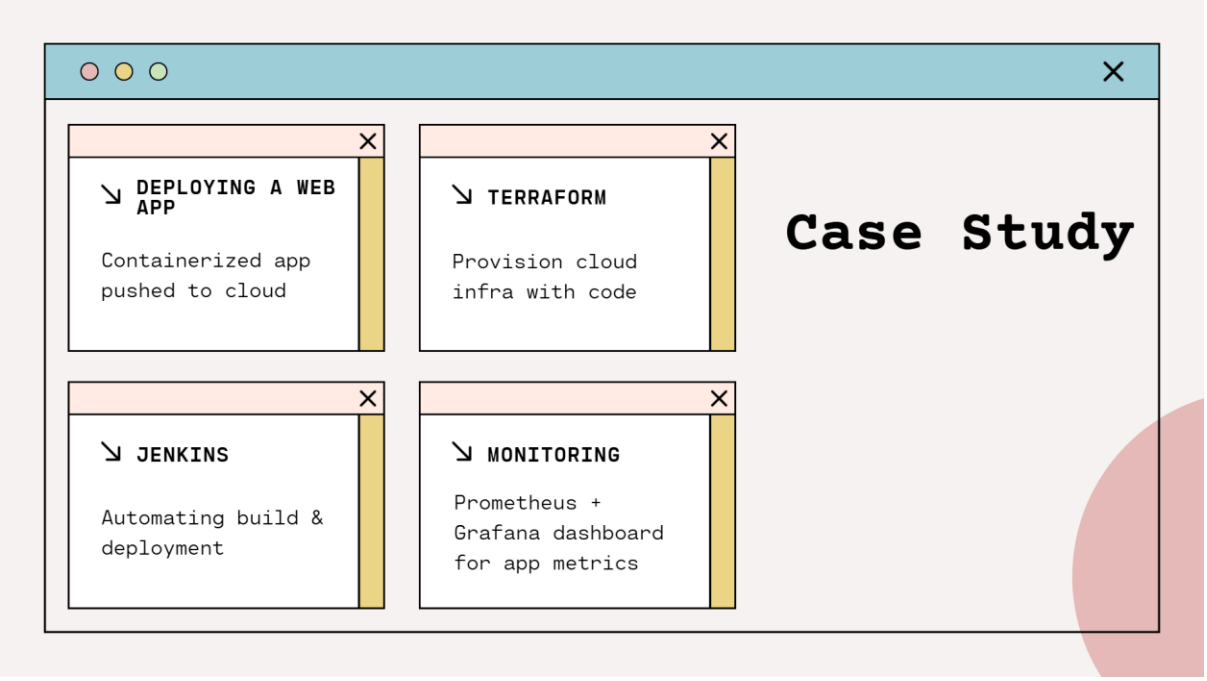

↘ WHAT IS IAC?
Infrastructure defined in code for repeatability & automation
↘ TERRAFORM PROVIDERS
AWS, Azure, GCP providers and authentication basics
↘ TERRAFORM WORKFLOW
init → plan → apply → destroy
↘ SIMPLE EXAMPLE
Provisioning an EC2 or GCP VM with Terraform.
Infrastruc ture as Code


INTEGRATING JENKINS WITH CLOUD
Using plugins for AWS/GCP deployment
PIPELINE STAGES
Build → Test → Package → Deploy → Monitor
SECRETS & CREDENTIALS
Managing cloud keys securely in Jenkins
END-TO-END FLOW
Automating Docker image build and deployment to cloud
CI/CD Pipeline Deployment to Cloud


PROMETHEUS
Metrics-based monitoring system
ARCHITECTURE
Architecture - Targets, exporters, time-series DB, queries
INTEGRATION
Integration with Applications - Exposing metrics endpoint
ALERTING
Alerting - Using Alertmanager for notifications
Monitoring with Prometheus


↘ GRAFANA
Dashboards for time-series monitoring
↘ DATA SOURCES
Connecting Data Sources - Linking Prometheus to Grafana
↘ DASHBOARDS
CPU, memory, request latency visualization
↘ ROLE IN DEVOPS
Proactive monitoring & reliability improvement
Visualiza-tion with Grafana


↘ ELK COMPONENTS
Elasticsearch, Logstash, Kibana
↘ LOGGING
Collecting logs from multiple services
↘ KIBANA
Searching & visualizing logs
↘ ELK VS PROMETHEUS/GRAFANA
Logs vs metrics distinction
Log Management with ELK Stack


↘ DEPLOYING A WEB APP
Containerized app pushed to cloud
↘ TERRAFORM
Provision cloud infra with code
Case Study
↘ JENKINS
Automating build & deployment
↘ MONITORING
Prometheus + Grafana dashboard for app metrics