SDE: System Design and Engineering

Lecture – 5

Introduction to
Cloud and Deployment

From Zero to Google: Architecting the Invisible Infrastructure

by

Aatiz Ghimire

Sections

- Cloud Computing & Native Infrastructure
- DevOps Paradigm & CI/CD Concepts
- Docker & Kubernetes Essentials
- Deployment Strategies
- Kubernetes Patterns & Cloud Anti-Patterns
- Design Topics

What is Cloud Native?

Cloud native refers to **designing, building,** and **operating** applications that fully exploit the **scalability, elasticity, and resilience** of cloud computing—across public, private, or hybrid clouds.

Why Cloud Native Matters

- Enables scalable and resilient systems
- Leverages containerization and orchestration
- Automates deployment and operations
- Aligns with agile and DevOps methodologies

Evolution of Architecture & Processes

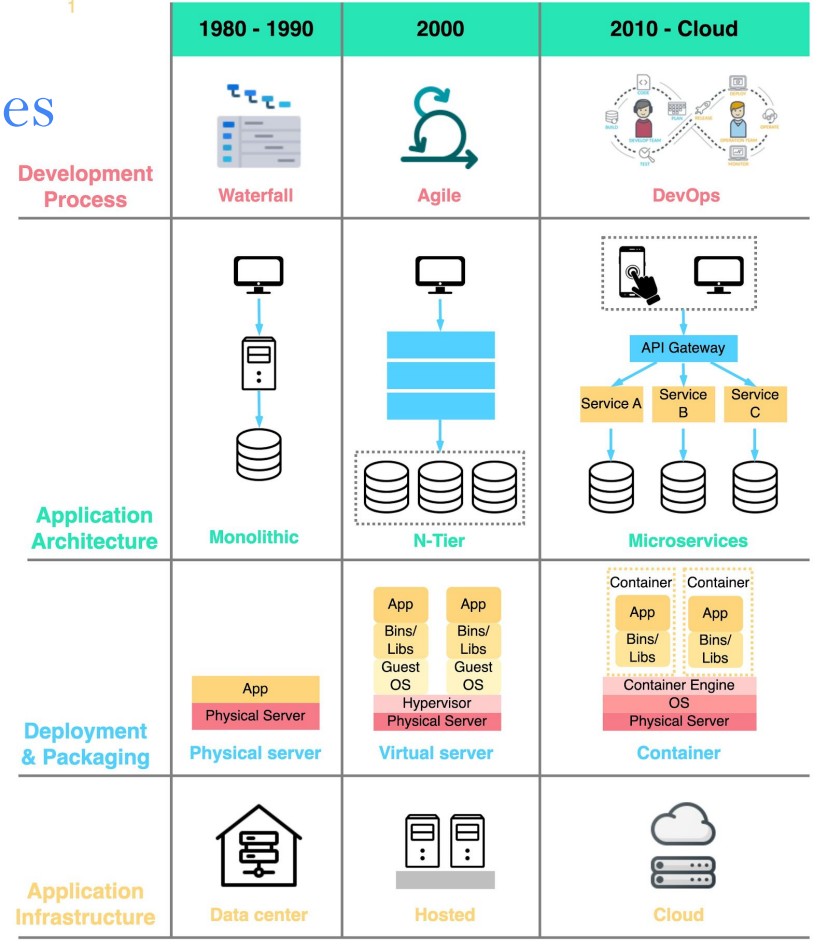
The 4 Pillars of Cloud Native

1. Development Process

- Transitioned from Waterfall → Agile → DevOps
- Focus on rapid iteration, automation, and feedback

2. Application Architecture

- Monoliths replaced by loosely coupled **microservices**
- Services are independently deployable and scalable
- Optimized for container environments



Evolution of Architecture & Processes

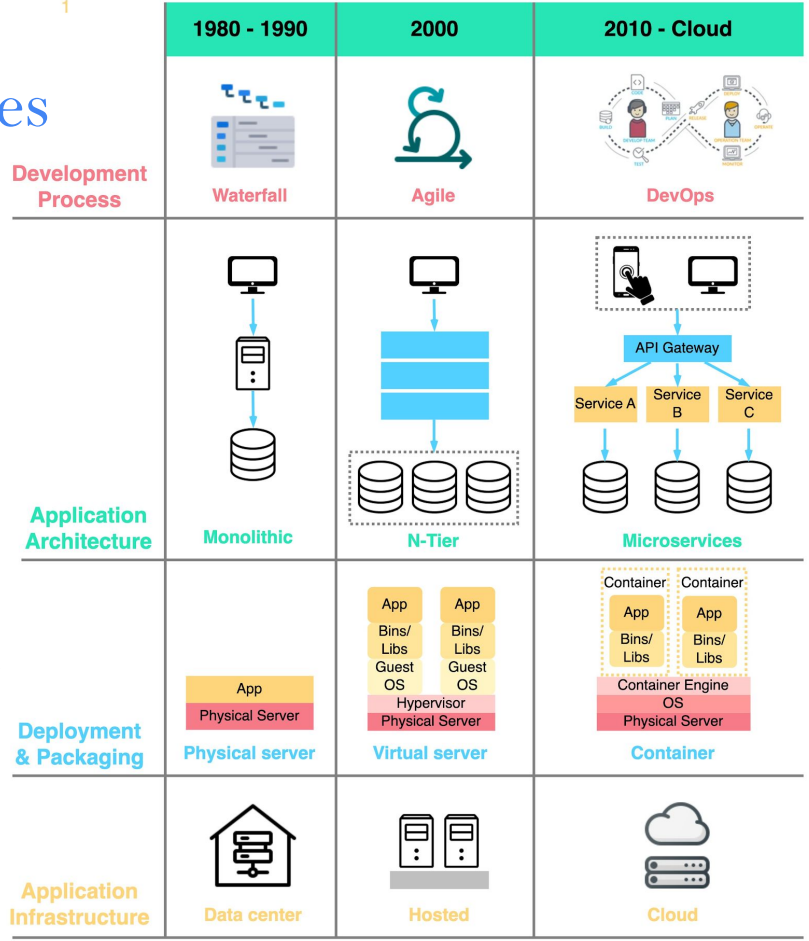
The 4 Pillars of Cloud Native

3. Packaging & Deployment

- From physical servers to **VMs**, now to **Containers**
- Docker images enable consistent, portable packaging
- Deployed via orchestrators (e.g., Kubernetes)

4. Application Infrastructure

- Cloud-native apps are deployed on **public, private, or hybrid clouds**
- Emphasize **horizontal scalability**, **fault-tolerance**, and **automation**
- Infrastructure-as-Code enables reproducible deployments



Terraform: Infrastructure as Code (IaC)

Terraform allows developers and operators to **define cloud infrastructure declaratively** in configuration files, which are then **automatically provisioned** through APIs.

The Terraform Workflow — Overview

Step 1 — Write Infrastructure as Code

- Define infrastructure in .tf files using HCL (HashiCorp Configuration Language)
- Use:
  - **Providers** (e.g., AWS, Azure, GCP, Kubernetes)
  - **Resources** (VMs, Networks, Databases)
  - **Modules**, **variables**, and **functions** for reuse

🧩 *Tip:* Use Terraform Registry for prebuilt modules

Terraform: Infrastructure as Code

2. Terraform Plan

- Command: terraform plan
- Compares:
  - **Desired state** (in code)
  - vs **Current state** (in state file)

- Outputs a preview of:
  - Additions
  - Deletions
  - Modifications

💡 Often integrated in **CI/CD pipelines** for safety & automation

Terraform: Infrastructure as Code

3. Terraform Apply

- Command: terraform apply
- Executes all necessary API calls to cloud providers
- Provisions, updates, or deletes cloud resources
- Updates the **Terraform state file**

🛠️ *Supports multiple providers and even Kubernetes*

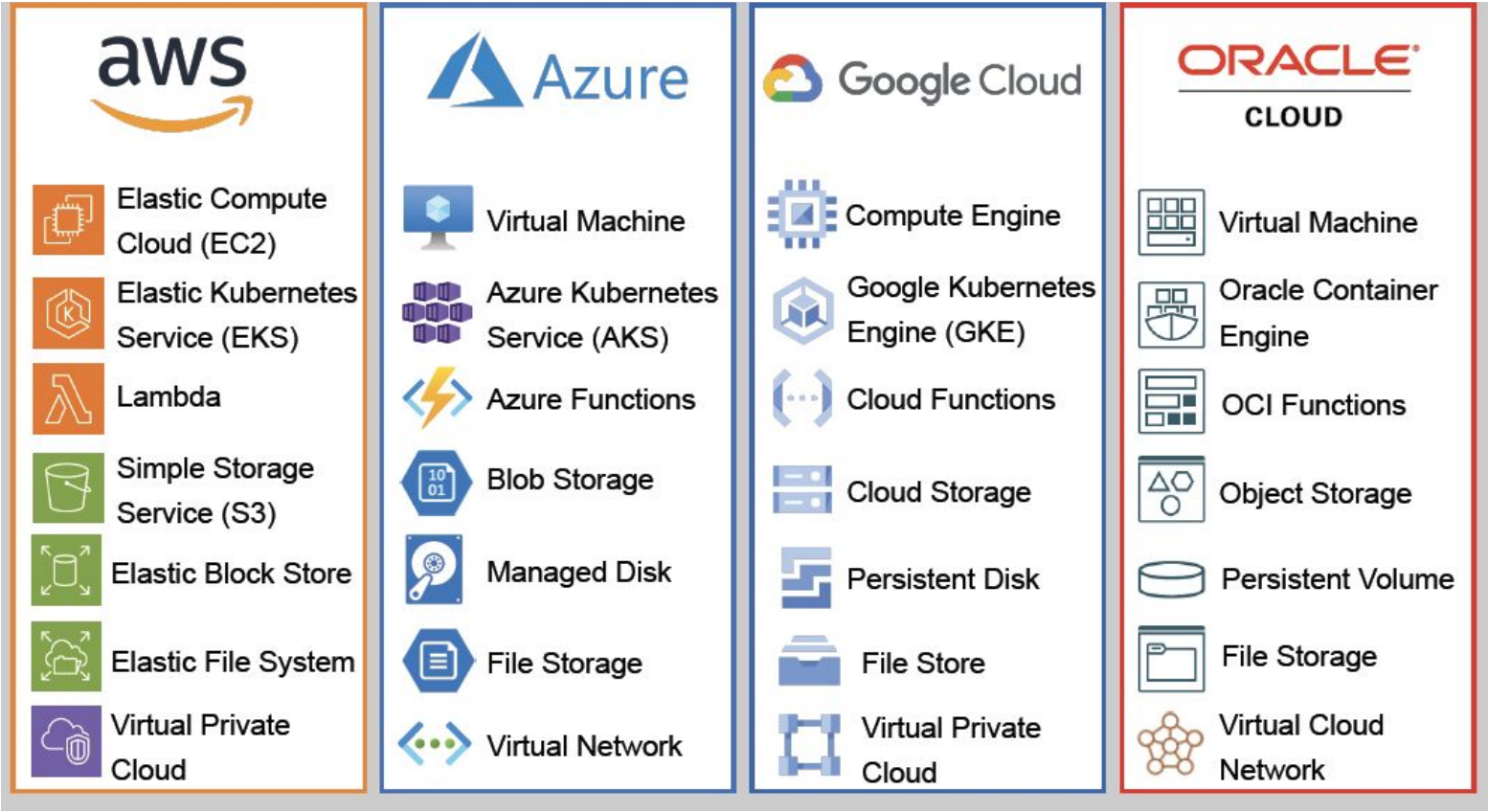
Terraform: Infrastructure as Code

4. Infrastructure Ready

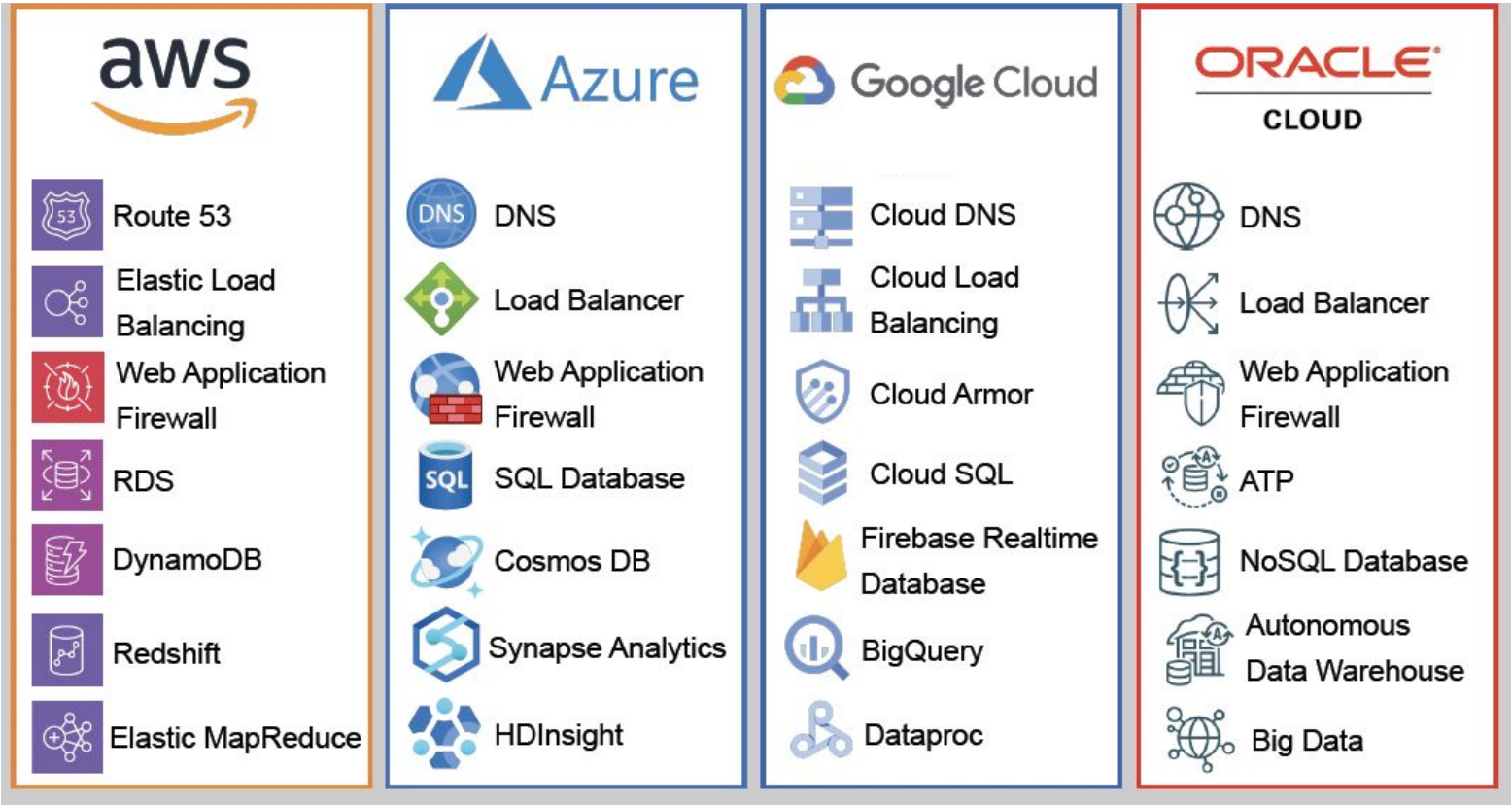
- **Terraform State File (terraform.tfstate)**
  - Tracks the real-world state of resources
  - Acts as **single source of truth**
  - Enables **collaboration**, **versioning**, and **rollbacks**

🔁 Future runs rely on the state file for drift detection and changes

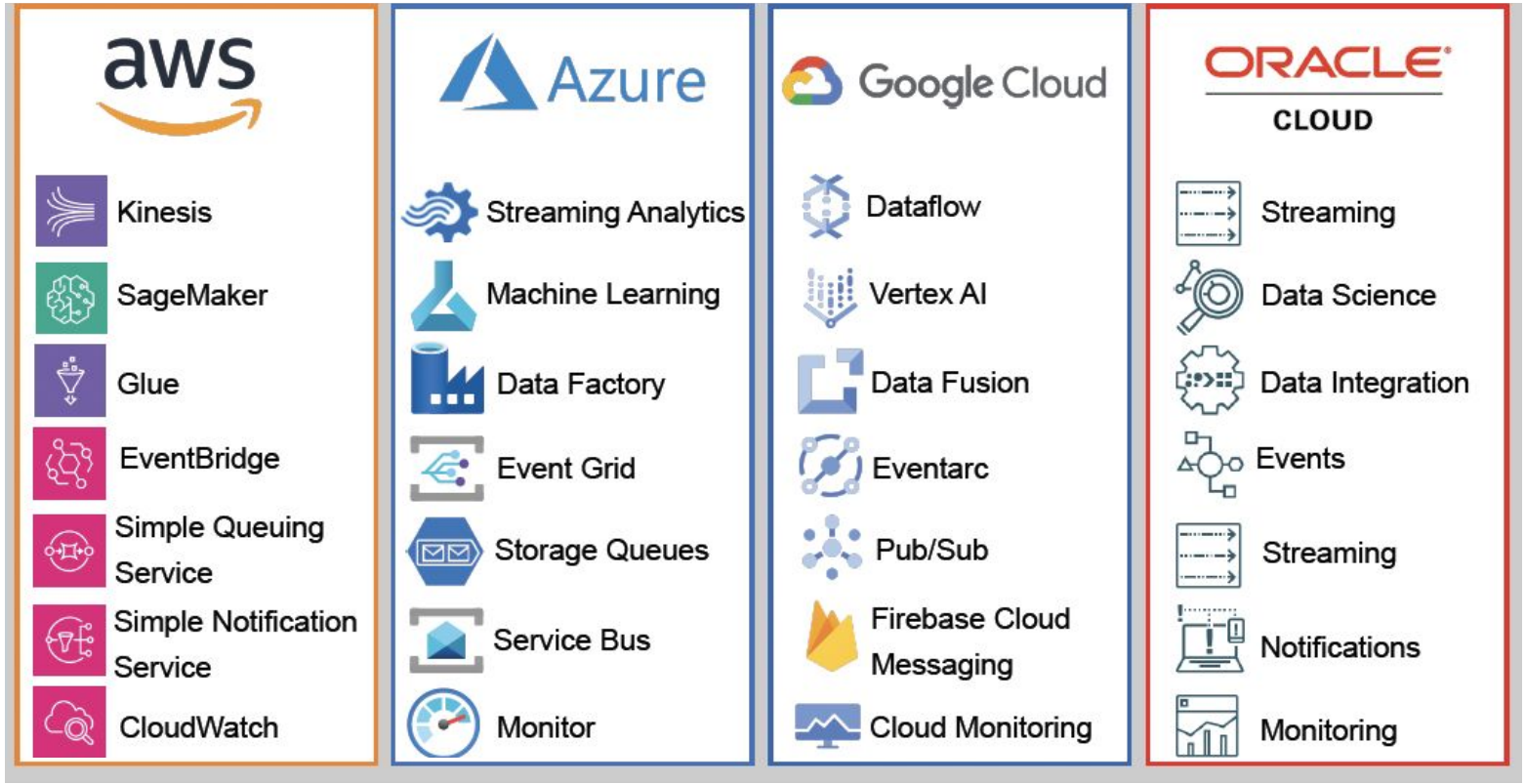
Cloud Provider Comparison



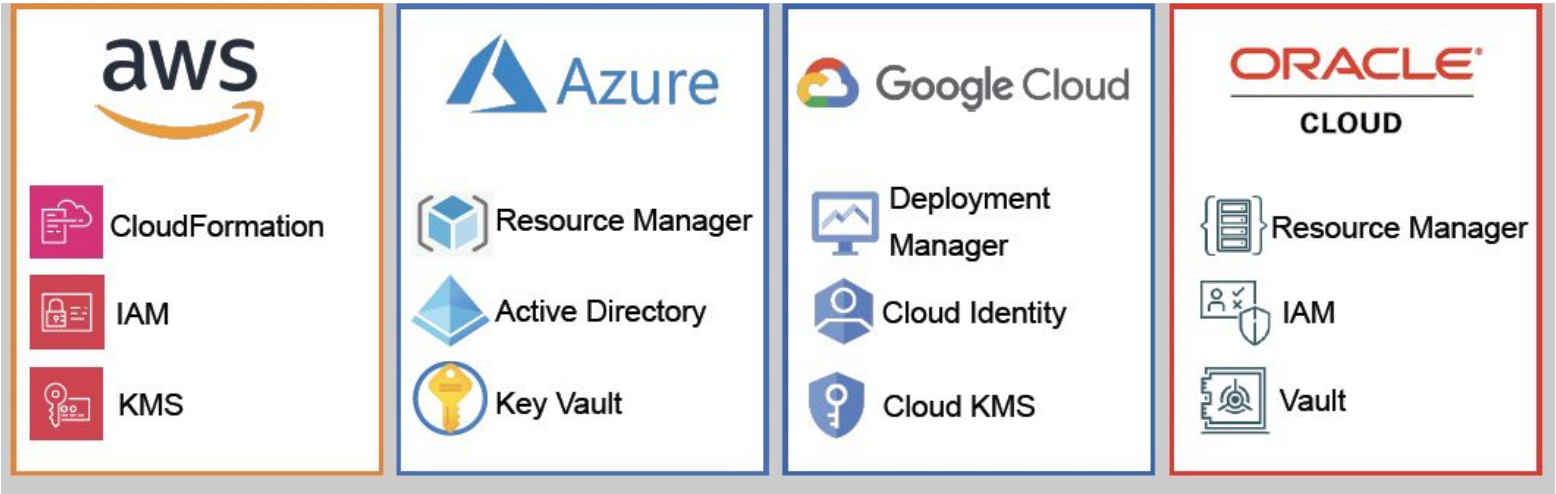
Cloud Provider Comparison



Cloud Provider Comparison



Cloud Provider Comparison



AWS Ecosystem & Services

Most Important AWS Services to Learn

Category	Key Services
Compute	EC2, Lambda, ECS, EKS
Storage	S3, EBS, Glacier
Database	RDS, DynamoDB, Aurora
Networking	VPC, ELB, Route 53
DevOps	CloudFormation, CodePipeline, CloudWatch
AI/ML	SageMaker, Rekognition
Security	IAM, KMS, Cognito

AWS Ecosystem & Services

Core AWS Network Components

Component	Description
VPC	Virtual network to launch resources
AZ	Redundant data centers in a region
IGW	Internet Gateway for external access
VGW	Virtual Private Gateway for VPN
VPC Peering	Direct traffic between VPCs
Transit Gateway	Hub to connect multiple VPCs/VPNs
Endpoints (Gateway/Interface)	Private connections to AWS services
PrivateLink	Secure SaaS access over private links

AWS Ecosystem & Services

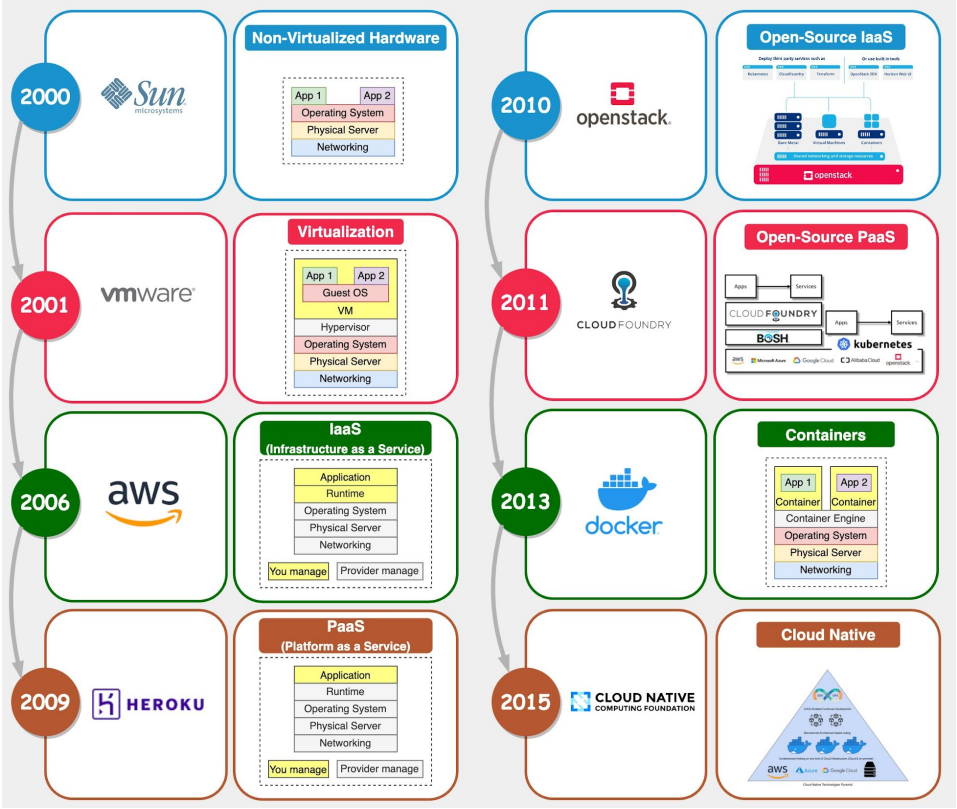
Network Connectivity Use-Cases

Scenario	AWS Service
Internet access	Internet Gateway
Remote worker VPN access	Client VPN Endpoint
On-premise to AWS VPN	Virtual Gateway
VPC-to-VPC communication	VPC Peering
Multi-VPC architectures	Transit Gateway
Private service access	VPC Endpoints / PrivateLink

Cloud Evolution & Cost

Cloud Evolution

- 2001 - VMWare - Virtualization via hypervisor
- 2006 - AWS - IaaS (Infrastructure as a Service)
- 2009 - Heroku - PaaS (Platform as a Service)
- 2010 - OpenStack - Open-source IaaS
- 2011 - Cloud Foundry - Open-source PaaS
- 2013 - Docker - Containers
- 2015 - CNCF (Cloud Native Computing Foundation) - Cloud Native



Cloud Evolution & Cost

Hidden Costs of the Cloud

Cloud services are easy to start and scale—but cost complexity can **quietly drain your budget** if not managed properly.

Area	Description
Free Tier Ambiguity	Not all services are fully "free"; exceeding limits incurs charges
Elastic IPs	Charged hourly after 5 IPs—even if unused
Load Balancers	Billed hourly + per GB of data transfer
EBS Volumes	Charged even if unattached or unused
EBS Snapshots	Persist after volume deletion unless explicitly removed

Cloud Evolution & Cost

Storage-Related Hidden Costs

Service	Hidden Charges
S3 Access	GET, LIST, PUT requests may cost more than storage itself
S3 Partial Uploads	Incomplete multipart uploads are billed by the part
EBS Snapshots	Orphaned snapshots remain billable

Data Transfer Costs

- **Data In (Upload to AWS):** Usually free
- **Data Out (Download from AWS):** Can be **very expensive**, especially across regions or to on-premises
- **VPC Peering / Transit Gateway:** Internal transfer isn't always free

📌 *Monitor outbound traffic regularly and optimize transfer patterns.*

Cloud Evolution & Cost

Cost Mitigation Strategies

Strategy	Description
Tagging & Budgets	Use AWS Tags + Budgets for cost visibility
Automated Cleanup	Clear orphaned volumes, snapshots, uploads
Right-Sizing	Monitor and adjust instance/storage sizes
Reserved Instances	Commit to long-term usage for lower rates
Cost Explorer	Visualize usage and spot anomalies

DevOps Roles & Culture

Modern software delivery is driven by **collaboration, automation, and reliability**.

These three disciplines emerged to address **increasing complexity** in building, deploying, and managing software at scale:

- DevOps (2009)
- SRE (Early 2000s)
- Platform Engineering (2010s+)

DevOps Roles & Culture

DevOps

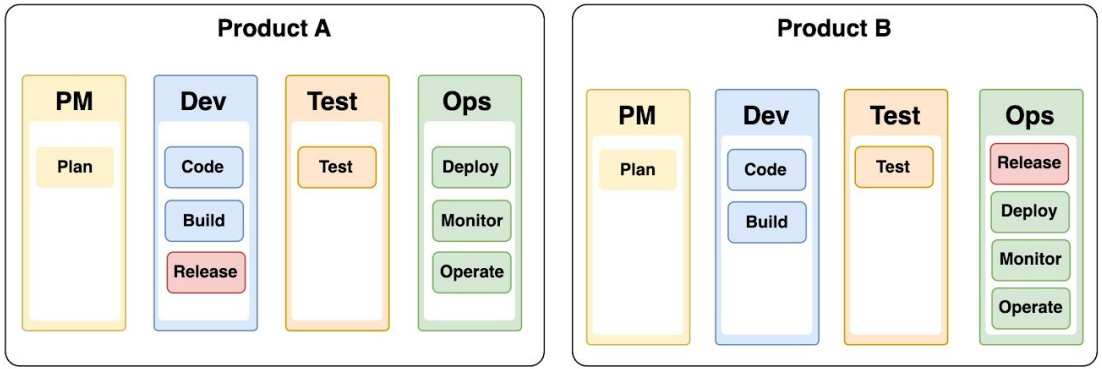
Founded: 2009 by **Patrick Debois** & **Andrew Shafer**
Context: Agile Conference

Key Idea:

- Break silos between *developers* and *operations*
- Foster *shared responsibility* and *continuous delivery*

📌 Core Practices:

- CI/CD pipelines
- Infrastructure as Code (IaC)
- Monitoring and alerting
- Culture of collaboration and iteration



DevOps Roles & Culture

SRE (Site Reliability Engineering)

Founded by: Google (Early 2000s)

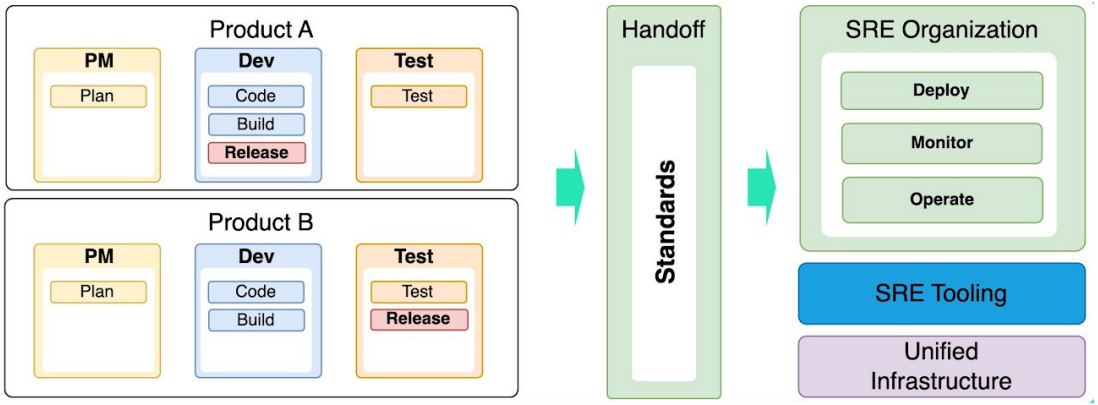
🧠 **Key Idea:**

- Apply *software engineering* to operations
- Maintain reliability while allowing fast deployments

📌 **Core Tools & Innovations:**

- **Borg** (cluster orchestration)
- **Monarch** (monitoring system)
- **SLIs, SLOs, Error Budgets**

👷 **SRE ≠ DevOps**, but both emphasize automation and resilience.



DevOps Roles & Culture

Platform Engineering

Emergence: 2010s+, derived from SRE and DevOps practices

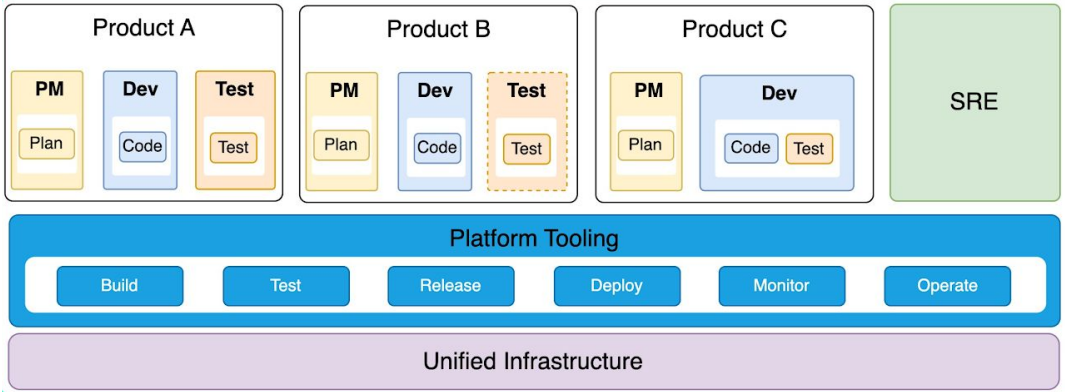
🧠 **Key Idea:**

- Build **internal platforms** that streamline the developer experience
- Abstract complexity of infrastructure and CI/CD

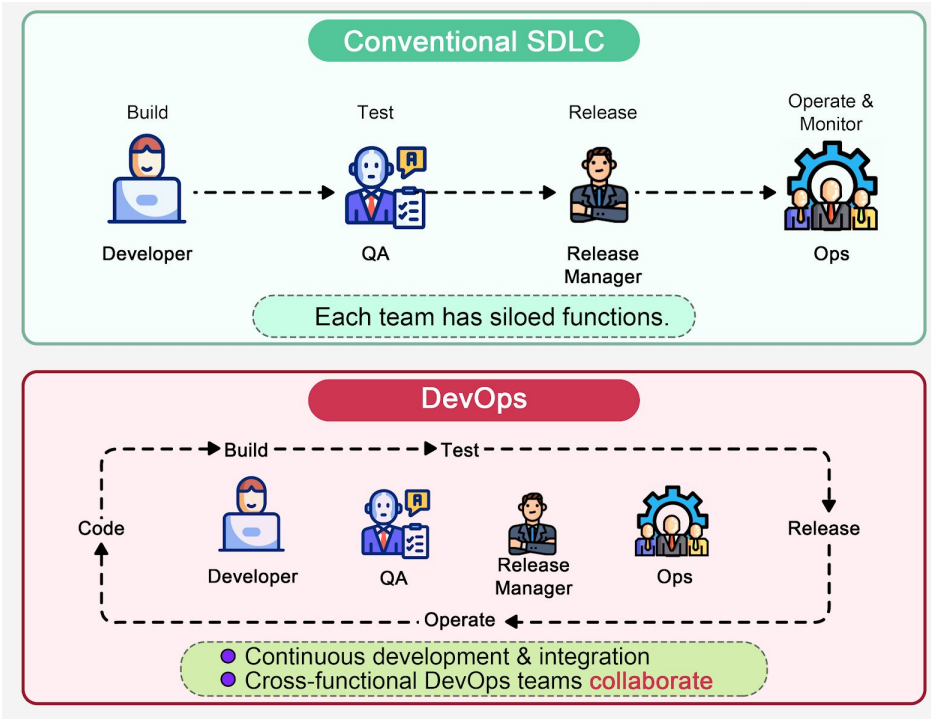
📌 **Core Focus Areas:**

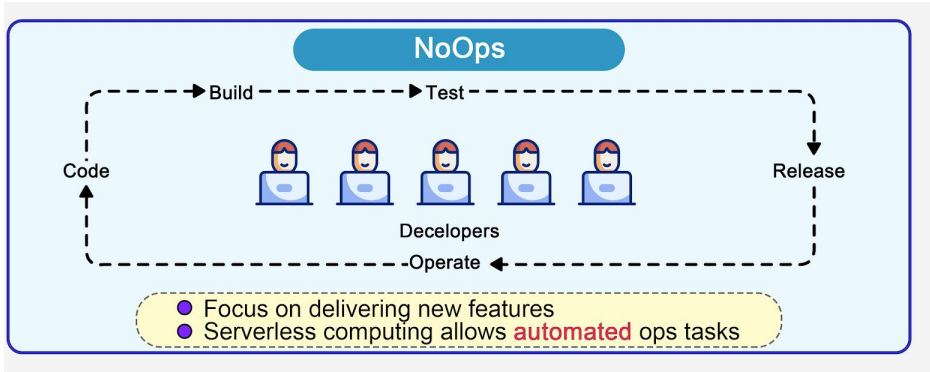
- Developer Portals (e.g., Backstage)
- Self-service pipelines
- Golden paths & templates
- Observability & policy-as-code

👥 *Platform engineers enable developer velocity at scale.*

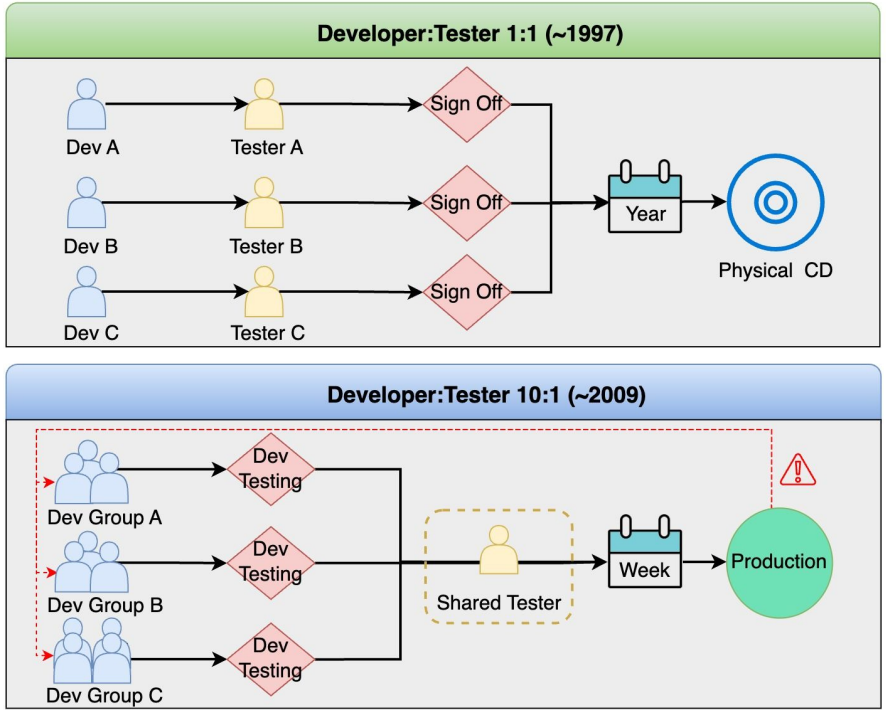


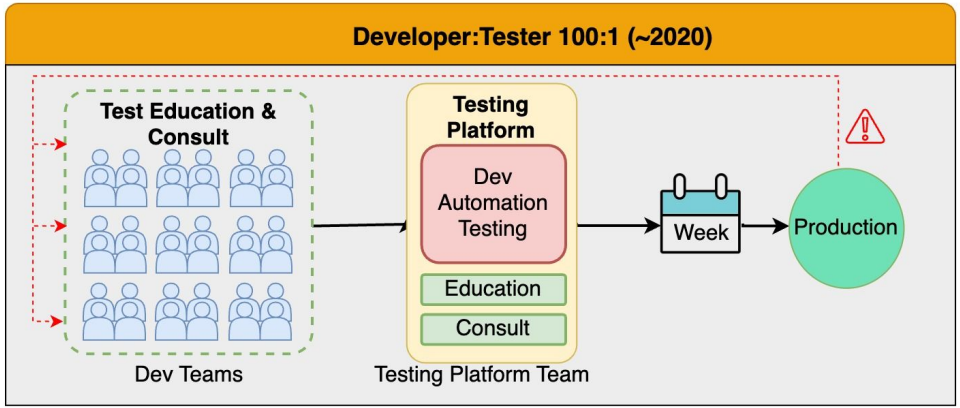
SDLC, DevOps and NoOps.





Paradigm Shift: Developer to Tester Ratio





CI/CD

- CI/CD stands for **Continuous Integration** and **Continuous Delivery/Deployment** — an automated process that connects **coding, testing, building, and deployment**.

🎯 **Goal:** Speed up software delivery with **automation and reliability**.

What's the Difference?

Aspect	Continuous Integration (CI)	Continuous Delivery/Deployment (CD)
Purpose	Detect errors early	Release code reliably & automatically
Trigger	On code commit	On test success or approval
Examples	Build, Unit Tests, Merge	Infra changes, Production deployment
Tools	Jenkins, GitHub Actions	ArgoCD, Spinnaker, AWS CodeDeploy

CI/CD in the SDLC

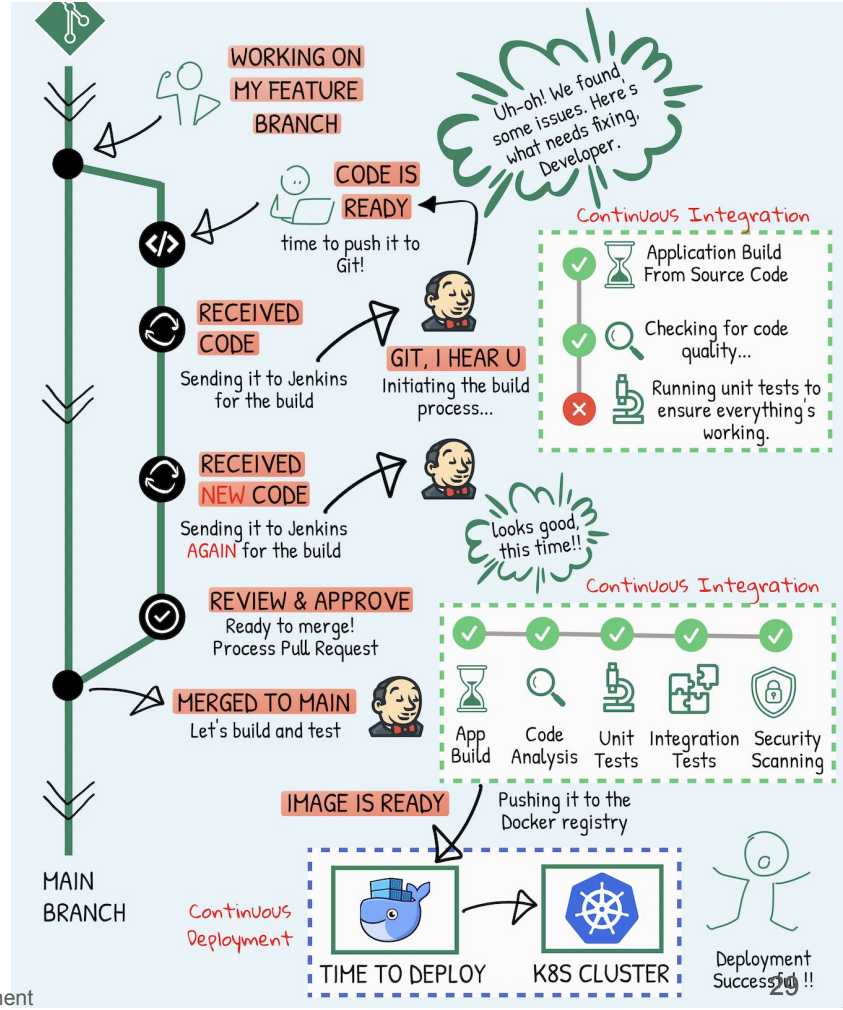
CI/CD automates the **Software Development Life Cycle (SDLC)** stages:

- **Development** → Commit code to version control
- **Build & Test** → Trigger CI pipeline
- **Staging** → Deploy artifacts
- **Production** → Release via CD pipeline
- **Maintenance** → Monitor and rollback if needed

📌 *Automation gives fast feedback, catches bugs early, and reduces risk.*

CI/CD Pipeline — Stage-by-Stage

1. Developer pushes code to **Git repository**
2. CI system **detects changes** (e.g., GitHub Actions)
3. Code is **compiled and tested**
   - Unit tests
   - Integration tests
   - End-to-end (E2E) tests
4. Test **results sent to developer**
5. If successful:
   - Deploy to **staging**
   - Run manual/automated QA
6. CD pipeline **promotes to production**



CI/CD Overview

Continuous Integration (CI)

- Encourages **frequent commits**
- Detects **integration bugs early**
- Ensures every change is **tested automatically**

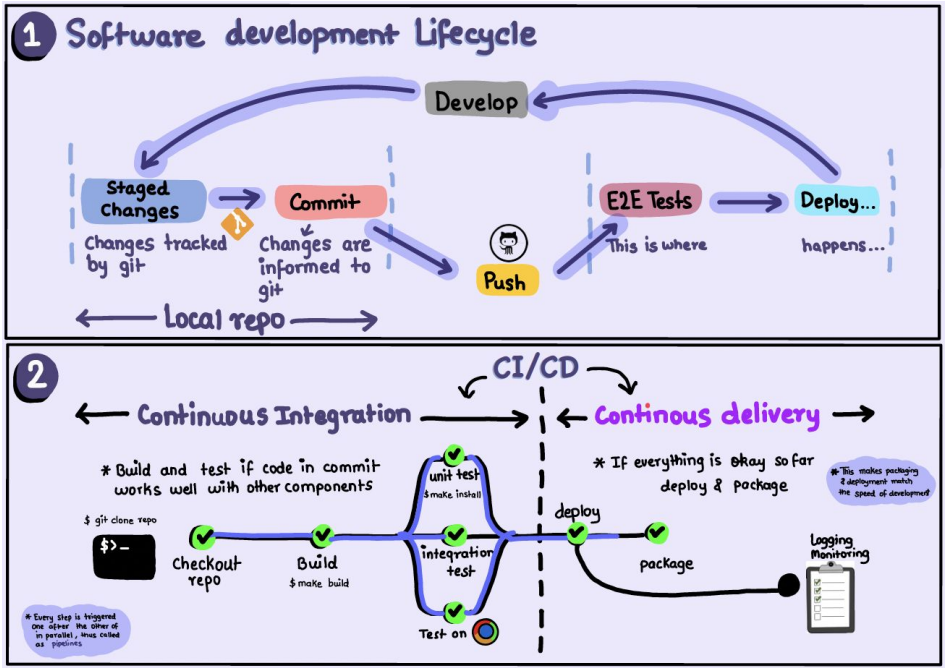
📌 *CI improves code quality and developer feedback loop.*

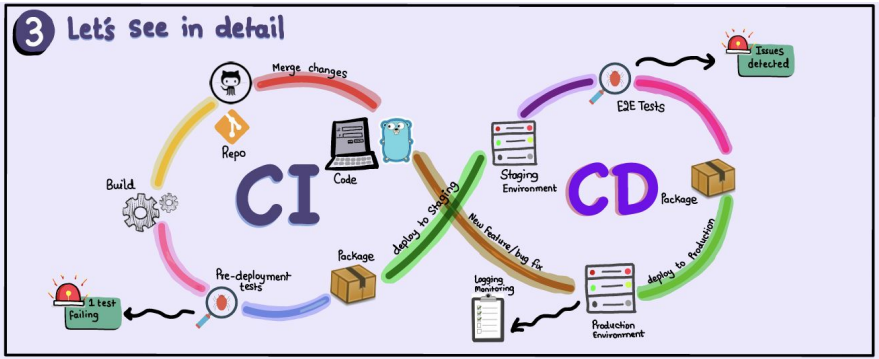
Continuous Delivery/Deployment (CD)

- Automates **infrastructure changes**
- Ensures app is **always deployable**
- Optional **approval steps** (Delivery) or **fully automated** (Deployment)

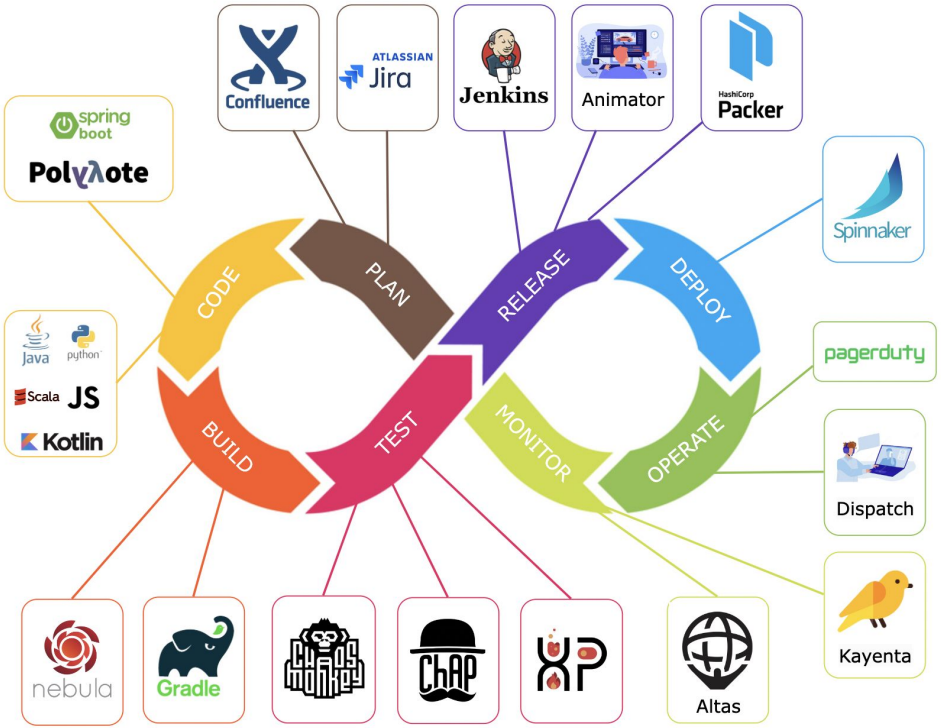
📌 *CD enables faster, safer releases with fewer outages.*

CI/CD Overview





Netflix Tech Stack - CI/CD Pipeline



- **Planning:** Netflix Engineering uses JIRA for planning and Confluence for documentation.
- **Coding:** Java is the primary programming language for the backend service, while other languages are used for different use cases.
- **Build:** Gradle is mainly used for building, and Gradle plugins are built to support various use cases.
- **Packaging:** Package and dependencies are packed into an Amazon Machine Image (AMI) for release.
- **Testing:** Testing emphasizes the production culture's focus on building chaos tools.
- **Deployment:** Netflix uses its self-built Spinnaker for canary rollout deployment.
- **Monitoring:** The monitoring metrics are centralized in Atlas, and Kayenta is used to detect anomalies.
- **Incident report:** Incidents are dispatched according to priority, and PagerDuty is used for incident handling.

Docker Essentials

Docker is a **containerization platform** that packages applications and their dependencies into lightweight, portable units called **containers**.

🎯 Enables:

- Consistency across environments
- Efficient DevOps workflows
- Fast, reproducible deployments

Docker Architecture – Core Components

Component	Description
Docker Client	CLI that interacts with the Docker daemon
Docker Daemon (Host)	Builds, runs, and manages containers
Docker Registry	Stores Docker images (e.g., Docker Hub)

Dockerfile → Image → Container

Concept	Description
Dockerfile	Instructions to build an image (base image, dependencies, commands)
Docker Image	Immutable package with code + runtime environment
Docker Container	Running instance of an image, isolated and secure

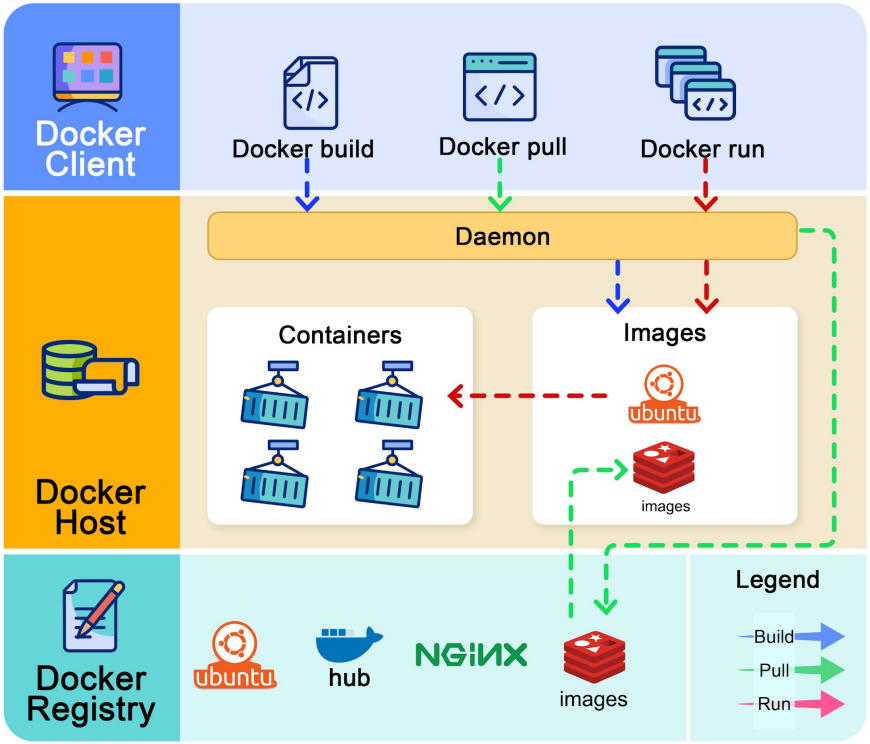
📌 *Dockerfiles enable repeatable, scripted image creation.*

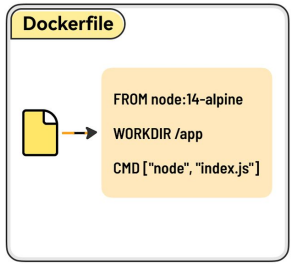
Docker Lifecycle — How It Works

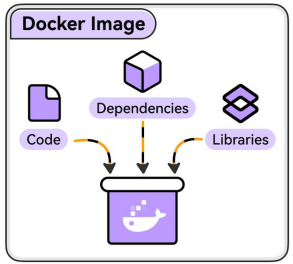
Command: docker run <image>

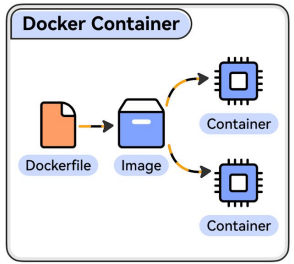
1. Docker Client sends request to Daemon
2. Daemon checks if image is local, else pulls from Registry
3. Daemon creates a **container**, allocates FS, connects network
4. Container starts and runs the app

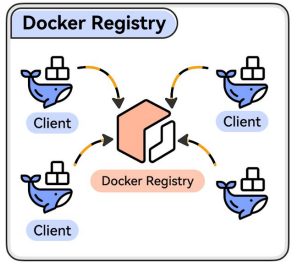
Docker Lifecycle — How It Works

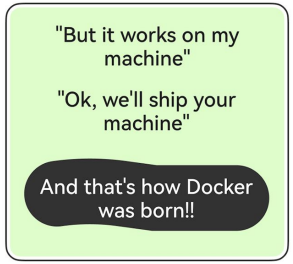


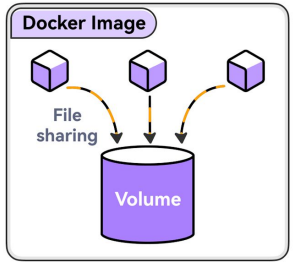


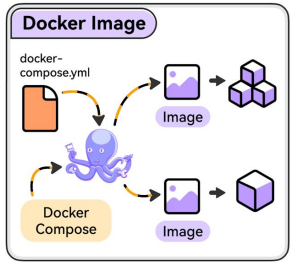


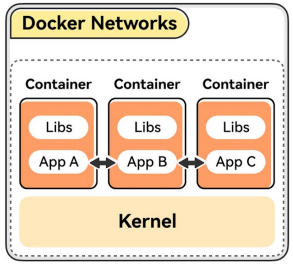


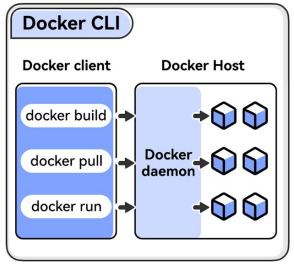












Extended Docker Concepts

Feature	Purpose
Docker Volumes	Persist and share data between containers
Docker Networks	Enable container communication and isolation
Docker Compose	Define and run multi-container applications
Docker CLI	Command-line interface for full Docker control

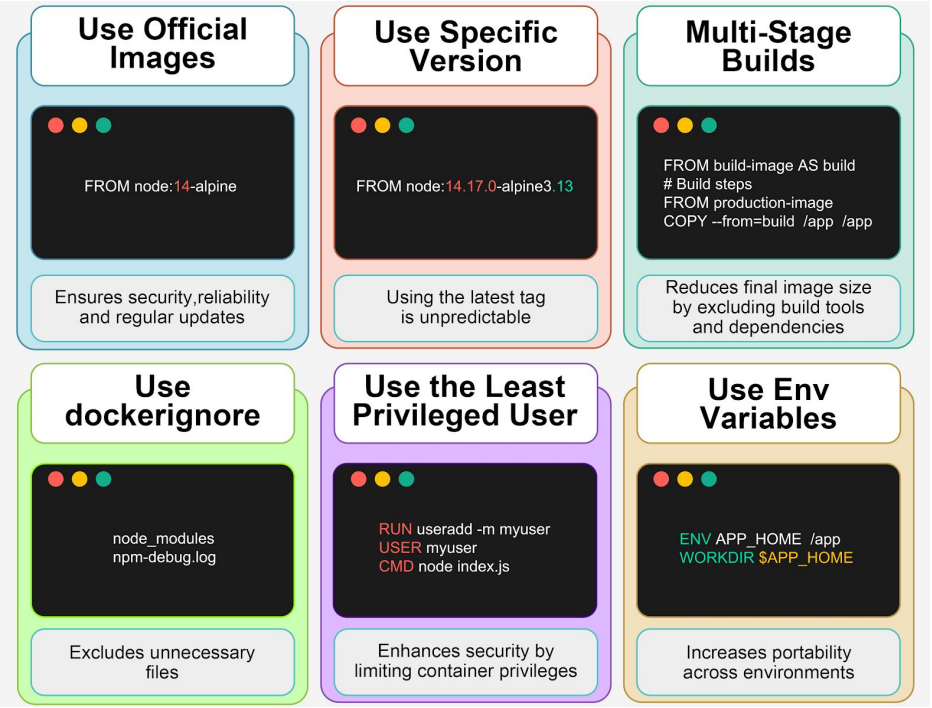
Example Dockerfile

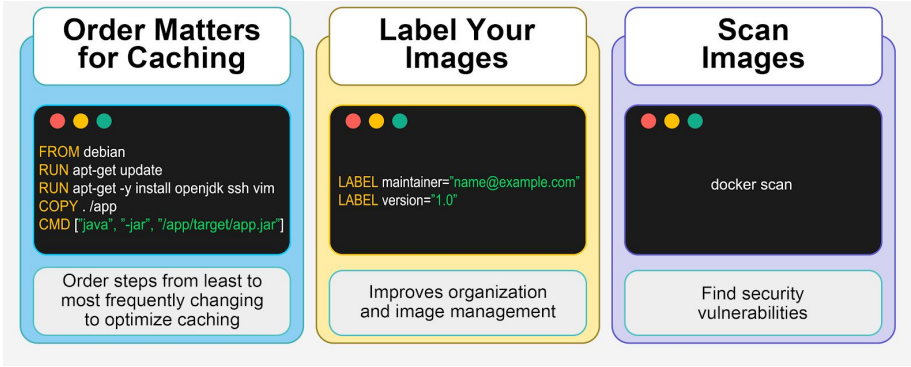
```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "app.py"]
```

Docker Best Practices

SN	Practice	Benefit
1	Use official images	Security, trust, regular updates
2	Pin specific image versions	Avoid unexpected changes
3	Use multi-stage builds	Reduce image size
4	Use .dockerignore	Faster builds, smaller context
5	Run as non-root	Improves container security
6	Use ENV variables	Flexibility across environments
7	Optimize layer order	Better caching, faster rebuilds
8	Label images	Metadata for tracking
9	Scan images	Detect security vulnerabilities early

Docker Best Practices





Why Use Docker?

- **Portability**: Works on any system with Docker engine
- **Isolation**: Avoids “it works on my machine” problems
- **Efficiency**: Lightweight, fast, reproducible environments
- **Scalability**: Easily integrates into CI/CD pipelines & Kubernetes

Kubernetes Essentials

- **Kubernetes (K8s)** is an open-source container orchestration system designed to automate deployment, scaling, and management of containerized applications.
- Inspired by **Google's internal Borg system**
- Now the **de facto standard** for container orchestration

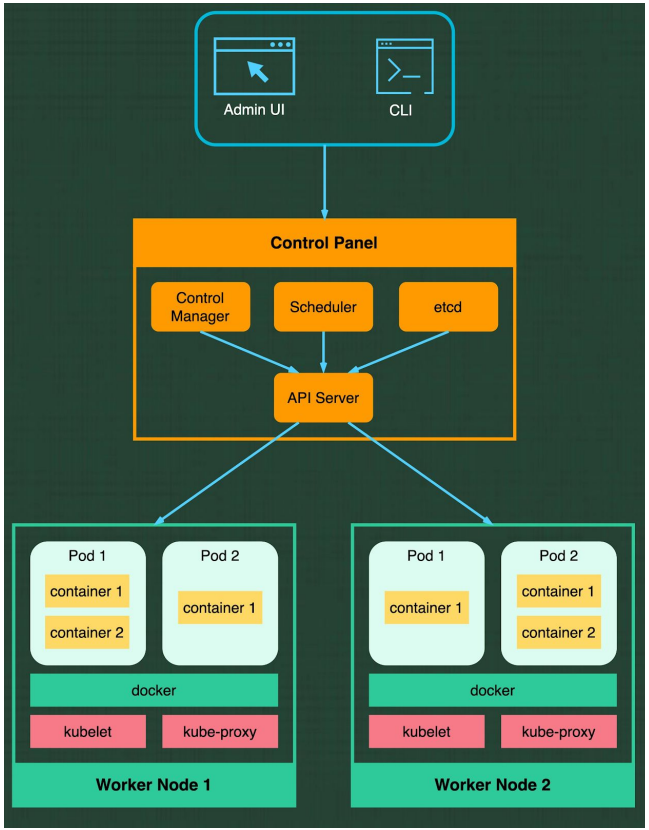
Kubernetes Cluster Design

- A **Kubernetes cluster** includes:
  - At least one **Control Plane**
  - One or more **Worker Nodes**

- **Pods** run on worker nodes
- The Control Plane manages the cluster's state

📌 *Supports fault tolerance and high availability in production.*

Components of K8s



Control Plane Components

Component	Role
API Server	Frontend to the control plane; all operations pass through here
Scheduler	Assigns new pods to appropriate nodes
Controller Manager	Runs built-in controllers (e.g., Node, Job, EndpointSlice)
etcd	Distributed key-value store for cluster state (high availability via Raft protocol)

Node Components

Component	Description
Pods	Smallest deployable unit; group of containers with shared IP
Kubelet	Agent that ensures containers are running in a Pod
kube-proxy	Maintains networking rules and forwards traffic to correct Pods

Kubernetes Essentials

A **Kubernetes Pod** is a group of one or more containers, with shared storage/network and a specification for how to run them.

All containers in a Pod:

- Share the **same IP address**
- Are scheduled on the **same node**

📌 *Pod is the smallest object Kubernetes can deploy.*

A **Kubernetes Service** is a stable abstraction that exposes a set of Pods to the network.

🎯 It enables:

- **Load balancing**
- **Automatic failover**
- **Internal/external access**

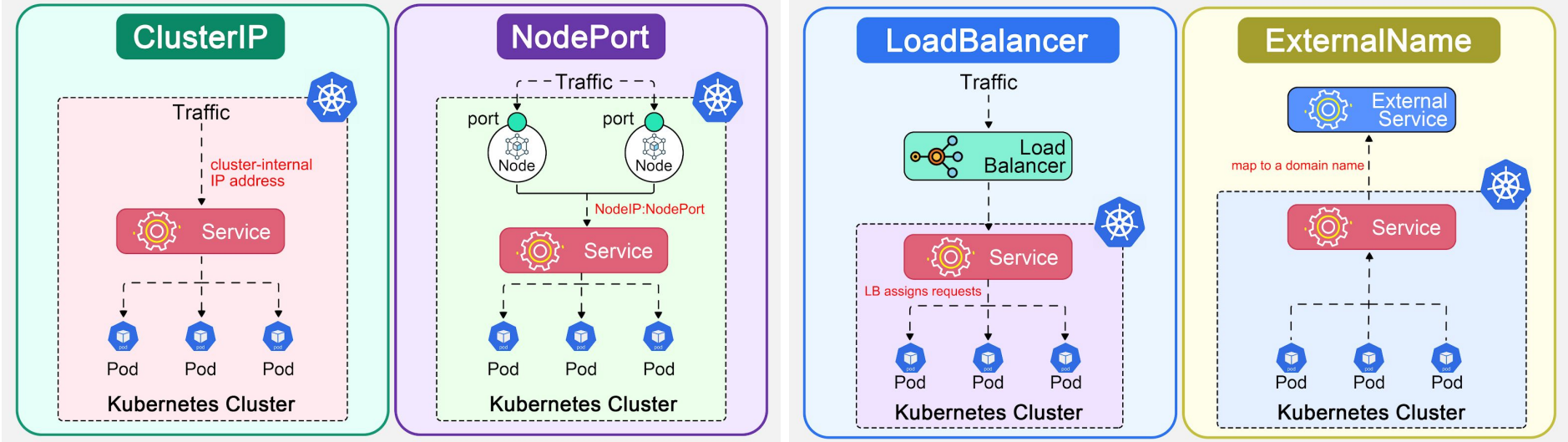
Top 4 Kubernetes Service Types

In Kubernetes, a Service is a method for exposing a network application in the cluster. We use a Service to make that set of Pods available on the network so that users can interact with it.

Service Type	Description
ClusterIP	Default type. Accessible only within the cluster.
NodePort	Exposes the service on each node's IP at a static port (NodeIP:NodePort).
Load Balancer	Integrates with external cloud load balancer (e.g., AWS ELB).
ExternalName	Maps the service to a DNS name (useful for external DBs).

📌 Use spec.type in your service YAML to configure.

Top 4 Kubernetes Service Types



Why K8s?

Kubernetes offers:

- Declarative infrastructure
- Scalable and self-healing orchestration
- Clear separation between **control plane** and **workload nodes**

Services abstract away:

- IP changes
- Pod lifecycles
- Networking complexity

Deployment Strategies

Why Are Deployments Risky?

- Software upgrades can break critical features
- Dependencies between services may fail under new versions
- Downtime, user impact, or irreversible data corruption can occur

🎯 **Goal**: Use strategies that minimize risk, allow testing, and support safe rollback

Strategy	Risk Level	Rollback Ease	Cost	Complexity
Multi-Service	🔴 High	🔴 Low	🟢 Low	🟢 Low
Blue-Green	🟢 Low	🟢 High	🔴 High	🟡 Medium
Canary	🟡 Medium	🟢 High	🟡 Medium	🟡 Medium
A/B Testing	🟡 Medium	🟢 Controlled	🟢 Low	🔴 High
Shadow	🟢 Very Low	🔴 Limited	🔴 High	🔴 High

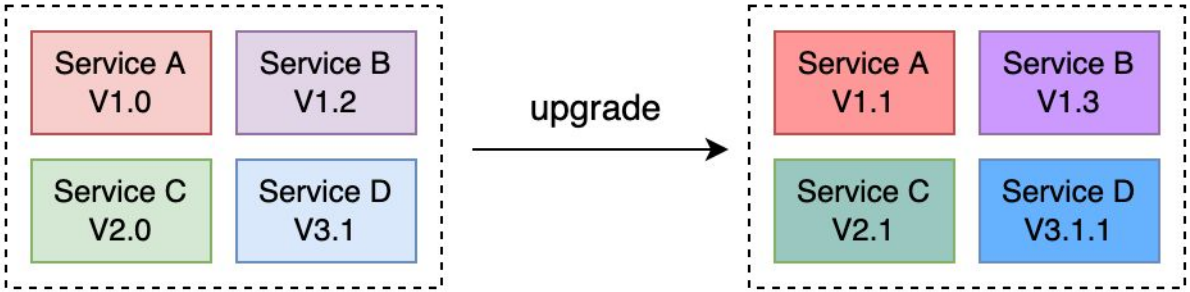
Key Deployment Strategies

1. Multi-Service Deployment

All services are upgraded **simultaneously**

✅ Easy to implement
❌ Hard to test dependencies
❌ Poor rollback support

📌 Best for: Small systems or tightly coupled services during early development

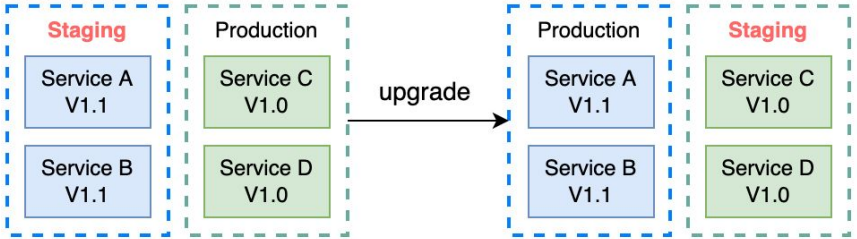


Key Deployment Strategies

2. Blue-Green Deployment

Maintain two environments (Blue = live, Green = new)

- Green is staged and tested
- Traffic switches to Green upon approval
- Easy rollback: just switch back to Blue



✅ No downtime
❌ Expensive (duplicate infrastructure)

📌 Best for: Critical applications with strict uptime requirements

Key Deployment Strategies

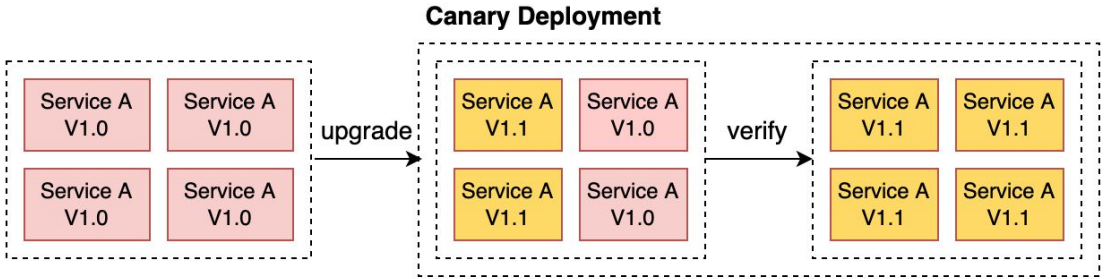
3. Canary Deployment

Gradually shift traffic to new version

- Start with a small % of users
- Monitor metrics (errors, latency, etc.)
- Increase exposure progressively

✅ Rollback only affects subset
❌ Requires **active monitoring** and traffic routing logic

📌 Best for: Staged rollouts in production



Key Deployment Strategies

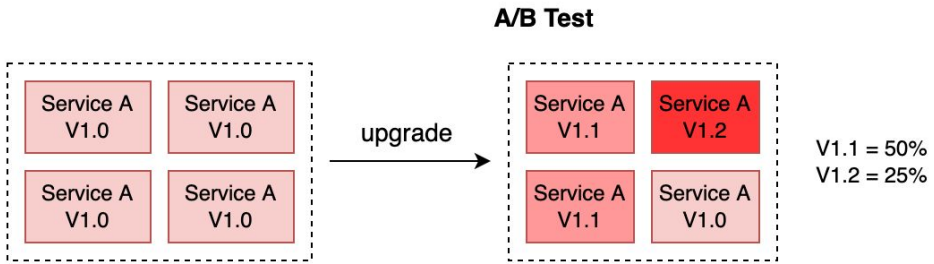
4. A/B Testing

Run multiple service versions concurrently for **experimentation**

- Traffic split between Version A and Version B
- Compare performance, UX, and KPIs

✅ Low cost, high UX insight
❌ Risk of unintended exposure
❌ Complex traffic management

📌 Best for: Product testing, UI experiments, or ML models



Key Deployment Strategies

5. Shadow Deployment

Clone live traffic to the new version **without exposing it to users**

- Used for **testing at scale** with real-world data
- No impact on actual users
- Needs **mock services** and infrastructure duplication

✅ Zero risk
❌ Complex and expensive

📌 Best for: Regression and performance validation at scale

Kubernetes Deployment Strategies

Strategy	Downtime	Description
Recreate	✅ Yes	Stops all old Pods, then starts new ones
Rolling Update	❌ No	Gradually replaces Pods to ensure uptime
Canary	❌ No	Updates subset of Pods/users, monitors impact
Blue-Green	❌ No	Two environments, switch over once verified
Shadow	❌ No	Live traffic mirrored to new version
A/B Testing	❌ N/A	Multiple versions served to distinct groups

Kubernetes Design Patterns

Why Design Patterns Matter

Kubernetes design patterns offer **reusable solutions** for common problems in container orchestration, including:

- Deployment behavior
- Pod structure
- Lifecycle management
- Infrastructure control

🎯 *Patterns improve scalability, maintainability, and automation.*

Kubernetes Design Patterns

Foundational Patterns

These are **universal principles** that ensure Kubernetes can manage the application effectively.

1. Health Probe Pattern

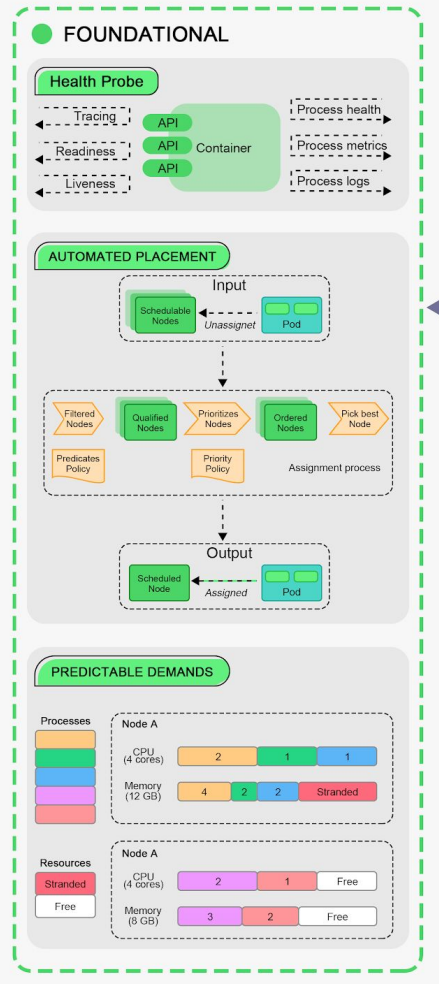
- Containers must expose **liveness** and **readiness** probes
- Enables K8s to auto-restart and load-balance intelligently

2. Predictable Demands Pattern

- Define **resource requests and limits** (CPU, memory)
- Ensures scheduler knows the container's needs

3. Automated Placement Pattern

- Kubernetes uses constraints (taints/tolerations, node affinity)
- Ensures optimized and rule-compliant pod placement



Kubernetes Design Patterns

Structural Patterns

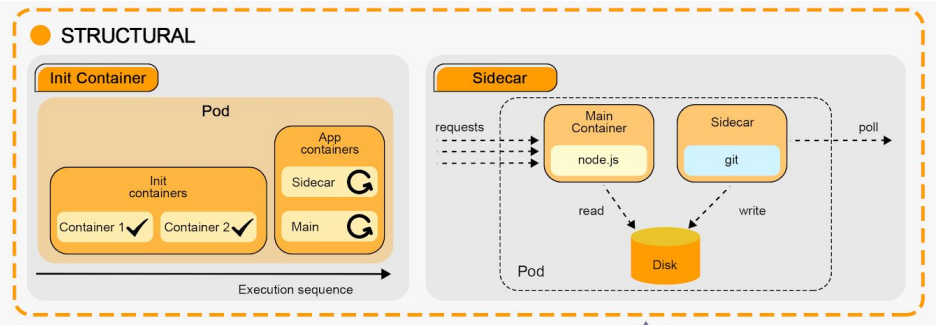
Focused on **organizing containers** inside Pods.

4. Init Container Pattern

- Runs setup logic **before** main container starts
- Ensures dependencies (e.g., configuration, DB migrations) are handled

5. Sidecar Pattern

- Runs a helper container **alongside** the main container
- Used for log shippers, proxies, or service mesh (e.g., Istio)



Kubernetes Design Patterns

Behavioral Patterns

Define how **pods behave across their lifecycle**.

6. Batch Job Pattern

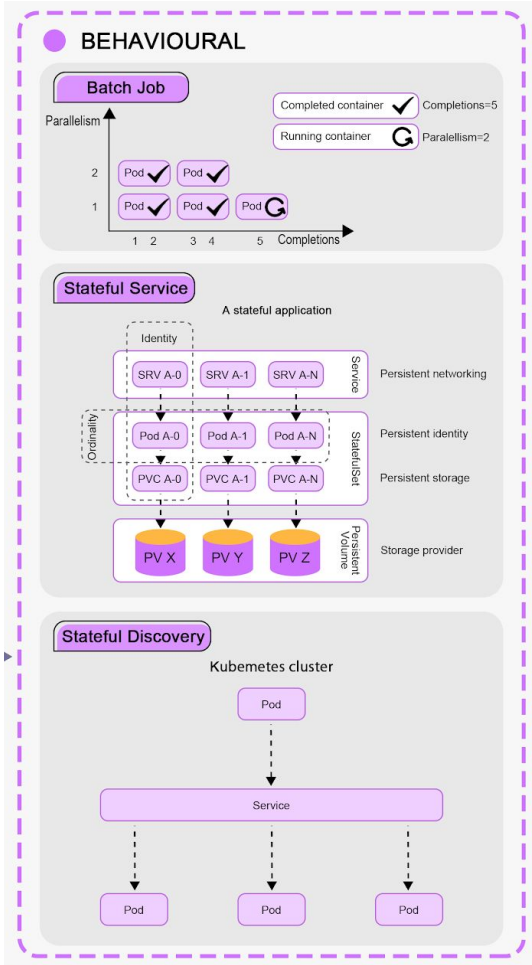
- Designed for **one-off tasks** (e.g., ETL jobs, cleanup scripts)
- Managed via Job or CronJob resources

7. Stateful Service Pattern

- Used for applications that require:
  - Stable network identity
  - Persistent storage
- Implemented using StatefulSet and PersistentVolume

8. Service Discovery Pattern

- Services are exposed internally using **DNS-based discovery**
- Enables Pods to discover and connect to each other via Service names



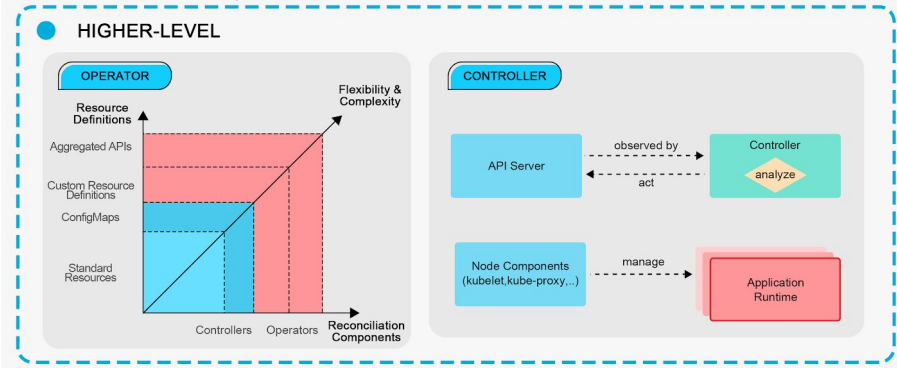
Kubernetes Design Patterns

Higher-Level Patterns

Enable advanced **automation and control** over application behavior.

9. Controller Pattern

- Observes current state, compares with desired state
- Reconciles differences using control loops
- Built into all core K8s resources (Deployment, ReplicaSet)



10. Operator Pattern

- Custom controller + custom resource definitions (CRDs)
- Encapsulates **domain-specific operational knowledge**
- Ideal for managing **databases**, **queues**, and **stateful apps**

📌 *"Automate the operator's brain" — for managing complex app lifecycles*

When to Apply These Patterns

Use Case	Pattern to Apply
Logging / Proxy / Monitoring	Sidecar
Data migration	Init Container
Long-running app	StatefulSet + Health Probes
Custom database automation	Operator Pattern
Cron-like scheduled job	Batch Job

Cloud Native Anti-Patterns

What Are Cloud Native Anti-Patterns?

Anti-patterns are **common but flawed practices** that hinder scalability, agility, and reliability in cloud-native systems.

🎯 *Identifying and avoiding these helps design more robust, scalable applications.*

Top Cloud Native Anti-Patterns

Anti-Pattern	Why It's Problematic
Monolithic Architecture	Hinders scalability, deployment speed, and modularity
Mutable Infrastructure	Causes config drift; harder to reproduce or scale
Ignoring Cost Optimization	Leads to budget overruns, unused resources
Large Container Images	Slower deployment; waste of resources
Inefficient DB Access	Performance bottlenecks, poor scaling
No CI/CD Pipelines	Error-prone, slow releases
Shared Resource Dependency	Creates tight coupling, hard to isolate issues
Stateful Components	Impedes auto-scaling and fault tolerance
Too Many Cloud Services Without Strategy	Operational complexity, cognitive overload

Best Practices to Replace Anti-Patterns

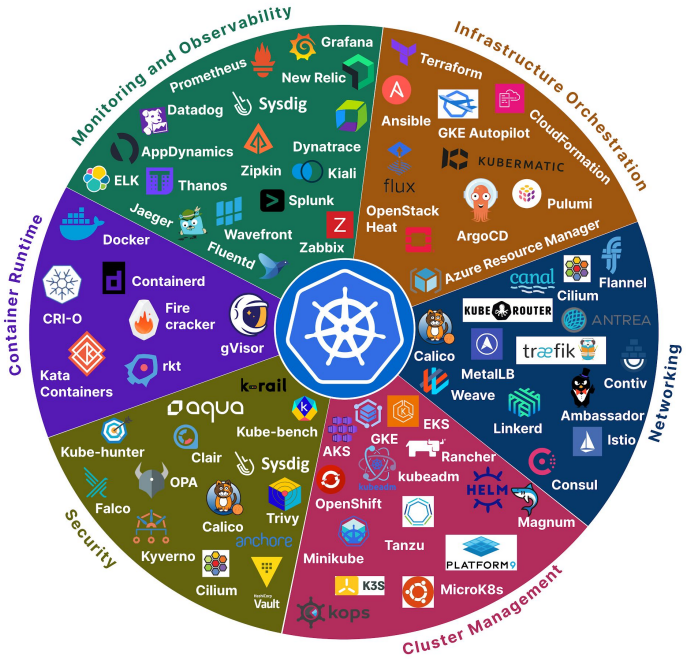
Instead of...	**Do This...**
Monolith	Break into Microservices
Mutable Infra	Use IaC (Terraform, Pulumi)
Manual Deployment	Implement CI/CD (GitHub Actions, Argo)
Bloated Images	Use Multi-Stage Builds, Alpine base
Shared DB	Use per-service storage abstraction
No Monitoring	Adopt observability stack (Prometheus, Grafana)

Kubernetes Tools Ecosystem Overview

Kubernetes is powerful—but needs **complementary tools** to be production-ready.

📌 Key Ecosystem Domains:

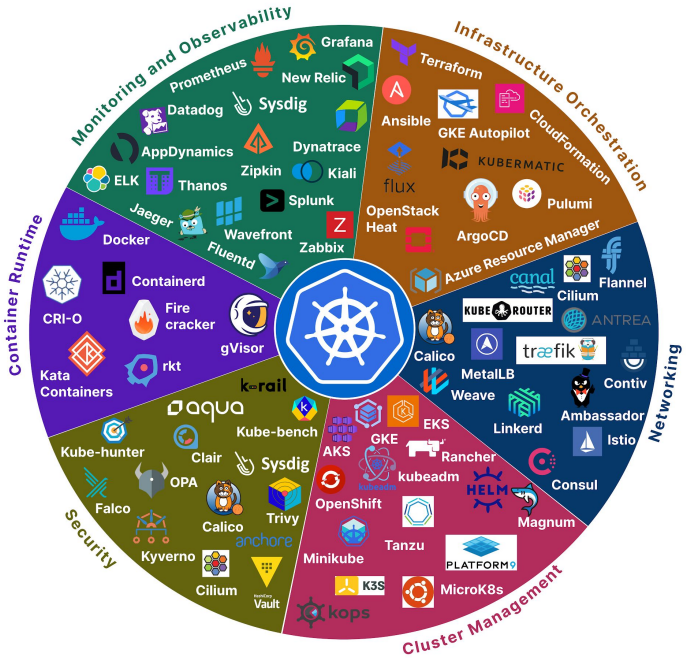
- Security
- Networking
- Container Runtime
- Monitoring & Observability
- Infrastructure & Cluster Management
- Continuous Deployment



🧠 *Choosing the right tools enhances performance, security, and agilit*

Kubernetes Tool Stack Wheel

Category	Tool Examples
Security	Kyverno, OPA/Gatekeeper
Networking	Calico, Cilium, Istio
Runtime	containerd, CRI-O
Observability	Prometheus, Grafana, Jaeger
Cluster Management	kubeadm, K3s, Rancher
Infra Orchestration	Helm, ArgoCD, Flux



Tooling Strategy Tips

✅ Keep your stack:

- **Minimal**: Avoid “tool sprawl”
- **Integrated**: Ensure tools work well together
- **Documented**: Clear policies for upgrades and observability
- **Monitored**: Every layer should emit metrics/logs

🔧 Use curated distributions like **Kubeflow**, **OpenShift**, or **Rancher** to simplify choices.

Kubernetes Deployment Summary

- Cloud native success depends on **avoiding anti-patterns** and **choosing the right tooling**
- Stay focused on:
  - **Immutable infrastructure**
  - **Automation (CI/CD)**
  - **Lightweight containers**
  - **Service-level isolation**
  - **Controlled tool adoption**

📌 *A robust Kubernetes deployment isn't just YAML—it's strategy.*

NGINX Popular?

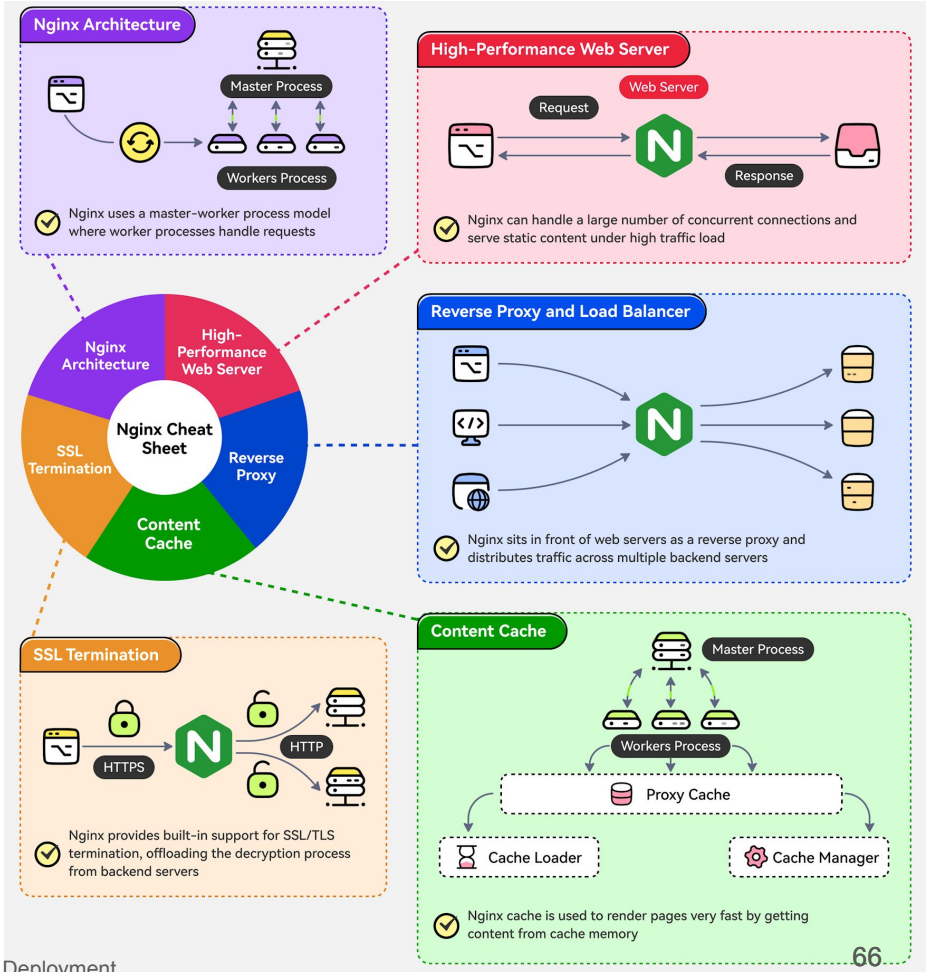
NGINX is a lightweight, event-driven server that acts as a **web server**, **reverse proxy**, **load balancer**, and **SSL terminator**—designed for high concurrency and low resource use.

Architecture:

- **Master Process**: Reads configs, manages workers
- **Worker Processes**: Handle connections using **non-blocking I/O**
- Follows an **event-driven model** for scalability and efficiency

Why NGINX?

- Handles thousands of concurrent connections
- Ideal for cloud-native apps, Kubernetes Ingress, and microservices gateways



Monorepo vs. Microrepo: Codebase Management Strategies

Monorepo: All code lives in one unified repository. Best for unified codebases with shared infra & strong review culture.

Microrepo: Each service/component has its own independent repository. Best for fast-moving teams & distributed service ownership

Aspect	Monorepo	Microrepo
Used by	Google, Meta, Airbnb, Linux, Windows	Amazon, Netflix, many microservice-based orgs
Structure	One repo, folders per service	One repo per service/component
Tooling	Bazel (Google), Buck (Meta), Nix, Lerna	Maven, Gradle, NPM, CMake
Dependencies	Shared & version-synced across all services	Scoped per repo, upgrade per team schedule
Code Quality	Global review standards, centralized control	Varies by team; governance can become inconsistent
Scaling	Scales dev standards; slower CI/CD without tooling	Scales org speed; may cause code fragmentation

Unique ID Generator in Scalable Systems

Unique ID Generator in Scalable Systems

Modern distributed systems (e.g., Twitter, Facebook) require **globally unique**, **time-sortable**, and **numerical** IDs for data integrity and performance.

Key Requirements

- **Globally Unique** (no collisions across machines/services)
- **Roughly Time-Ordered** (sortable for logs/events)
- **Numerical Only** (optimized for indexing and storage)
- **64-bit Format** (efficient, compact)
- **High Throughput, Low Latency** (generate millions/sec)

Popular ID Generation Strategies

Method	Description
UUID (v1/v4)	Universally unique, not time-sortable, 128-bit
Snowflake (Twitter)	64-bit ID with timestamp, machine ID, and sequence
ULID	Lexicographically sortable, 128-bit
KSUID	K-Sortable ID with embedded timestamp
MongoDB ObjectID	12-byte (96-bit), includes timestamp & machine ID

Best Practice:
Use **Snowflake-like 64-bit IDs** in high-scale systems where **ordering + uniqueness** matter (e.g., posts, events, logs).

Thank you!
Any Questions?