Bespoke Cloud Technology

Introduction to Cloud and the Cloud Native Model

What Is Cloud Computing?

- On-demand resources
- Elastic capacity
- Shared responsibility model
- API-driven infrastructure

On-demand resources

What it means:

- Infrastructure resources (compute, storage, network, managed services) are:
  - provisioned **automatically**
  - available in **seconds or minutes**
  - created **without human interaction** from the provider side

On-demand resources

Security / audit perspective:

- No manual approval does not mean no control
- Control shifts to:
  - IAM
  - policies
  - guardrails

On-demand resources

Security / audit perspective:

- IAM roles and permissions
- Cloud audit logs (API calls)
- Infrastructure-as-Code repositories (Terraform, CloudFormation)

Elastic capacity

What it means:

- Systems can:
  - scale up and down automatically
  - react dynamically to load
  - avoid fixed, static capacity

Elastic capacity

Security implications:

- Risk assessment changes:
  - not how many servers exist
  - but how scaling is controlled
- Common risks:
  - uncontrolled cost growth
  - amplification of DoS attacks
  - missing quotas or limits

Elastic capacity

Typical audit evidence:

- Autoscaling configurations
- Resource quotas and limits
- Budget alerts and monitoring rules
- Kubernetes HPA / cloud autoscaling policies

Shared responsibility model

What it means:

- Security responsibilities are shared between:
  - the cloud provider
  - the customer
- The exact boundary depends on:
  - IaaS vs PaaS vs SaaS
  - the specific service used

Shared responsibility model

Security / audit perspective:

- One of the most common audit failures:

  “We assumed the provider was responsible for this.”

- Audits must clearly define:
  - what is out of scope
  - what remains customer responsibility

Shared responsibility model

- Typical audit evidence:
  - Responsibility matrices
  - Provider security documentation
  - Architecture and data-flow diagrams

API-driven infrastructure

What it means:

- Everything in the cloud is:
  - created via APIs
  - managed via APIs
  - logged via APIs

API-driven infrastructure

Security implications:

- IAM becomes the primary security perimeter
- API abuse is a real attack vector
- Service accounts, tokens, and workload identities are critical assets

API-driven infrastructure

Audit value (very important):

- Full traceability:
  - who did what
  - when
  - from where
- Enables:
  - incident reconstruction
  - forensic analysis
  - policy enforcement validation

API-driven infrastructure

Typical audit evidence:

- Cloud audit logs
- Service account usage logs
- API call history
- GitOps and CI/CD pipeline logs

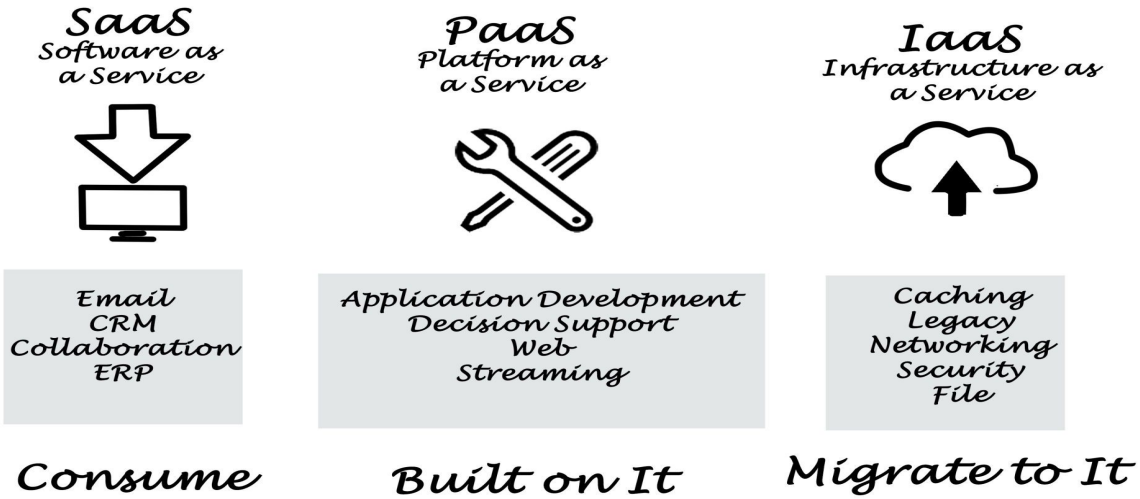
Cloud Provider Tools for Audit Evidence

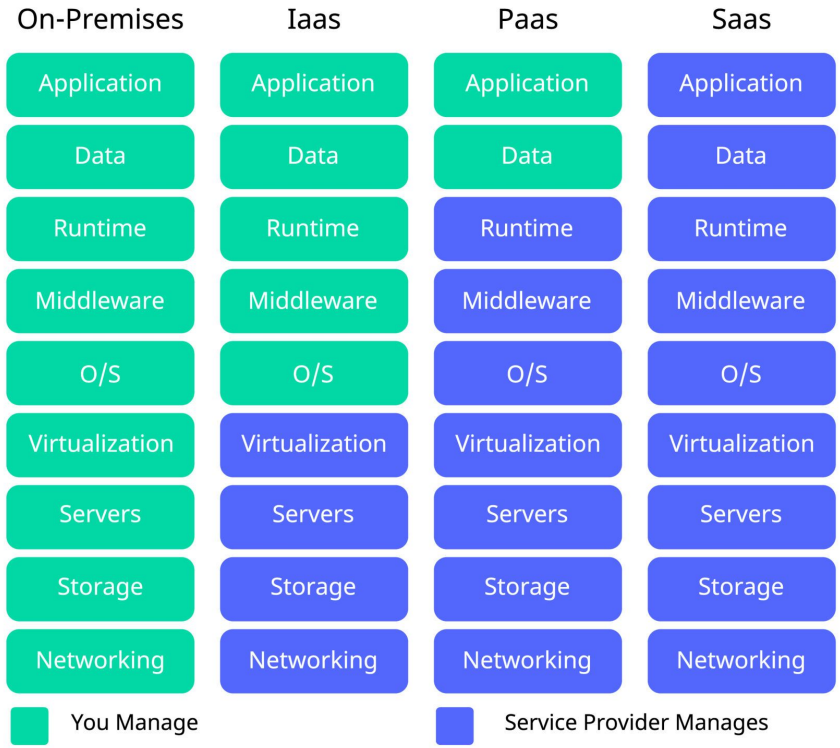
Key Areas and Native Tools for AWS, Google Cloud, and Azure

Audit Evidence Category	What Auditors Verify	aws	Google Cloud	Azure
Identity & Access (WHO)	Roles, Least Privilege	IAM, Access Analyzer	Cloud IAM, Org Policies	Azure AD, RBAC
API Activity (WHAT / WHEN)	API Call Logs, Admin Actions	CloudTrail	Cloud Audit Logs	Activity Logs
Configuration State (HOW IT IS SET)	Drift, Compliance	AWS Config	Config Validator	Azure Policy
Change Control (HOW IT CHANGED)	IaC, Versioned Changes	CloudFormation	Deployment Manager	ARM / Bicep
Security Posture (IS IT SAFE)	Threats, Misconfigurations	Security Hub, GuardDuty	Security Command Center	Defender for Cloud
Usage & Limits (HOW MUCH)	Quotas, Budgets	Service Quotas, Budgets	Quotas, Budget Alerts	Quotas, Cost Mgmt
Logs & Forensics (WHAT HAPPENED)	Log Data, Retention	CloudWatch Logs	Cloud Logging	Log Analytics

IaaS vs PaaS vs SaaS

- Different responsibility models
- Different audit controls



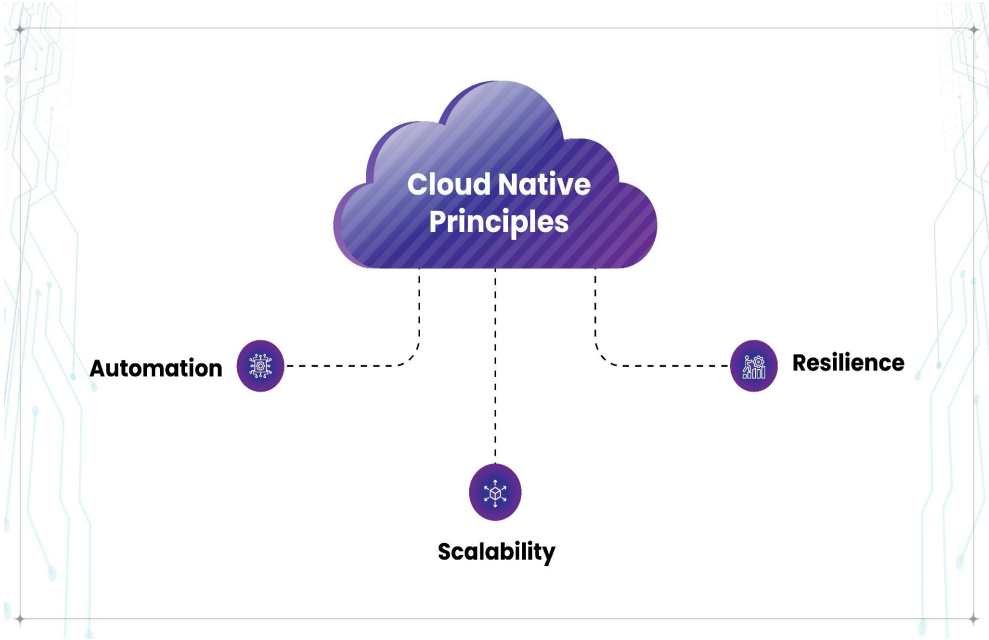

On-Premises
Iaas
Paas
Saas
Application
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking
You Manage
Service Provider Manages

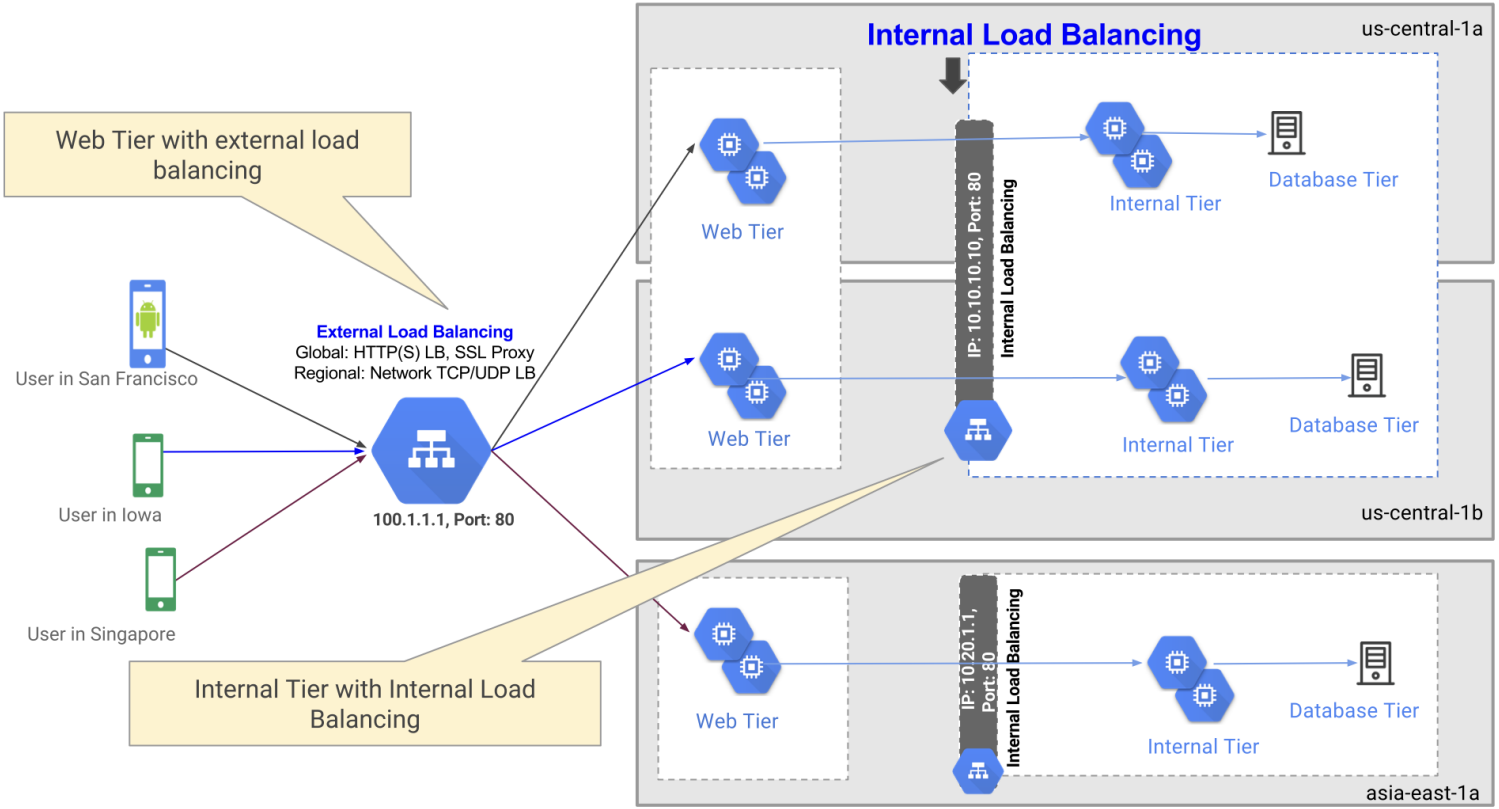
Cloud-Native Characteristics

- Immutable infrastructure
- Automation by default
- Horizontal scaling
- API-first design

Scalability, Resilience, Automation

- Failure is expected
- Self-healing systems
- Scaling without human approval




Internal Load Balancing
us-central-1a
Web Tier with external load balancing
Web Tier
Internal Tier
Database Tier
IP: 10.10.10.10, Port: 80
Internal Load Balancing
External Load Balancing
Global: HTTP(S) LB, SSL Proxy
Regional: Network TCP/UDP LB
User in San Francisco
User in Iowa
User in Singapore
100.1.1.1, Port: 80
Web Tier
Internal Tier
Database Tier
us-central-1b
Internal Tier with Internal Load Balancing
IP: 10.20.1.1, Port: 80
Internal Load Balancing
Web Tier
Internal Tier
Database Tier
asia-east-1a

Major Cloud Providers

CLOUD SERVICES	aws		
Virtual Servers	Instances	VM Instances	VMs
Platform-as-a-Service	Elastic Beanstalk	App Engine	Cloud Services
Serverless Computing	Lambda	Cloud Functions	Azure Functions
Docker Management	ECS	Container Engine	Container Service
Kubernetes Management	EKS	Kubernetes Engine	Kubernetes Service
Object Storage	S3	Cloud Storage	Block Blob
Archive Storage	Glacier	Coldline	Archive Storage
File Storage	EFS	ZFS / Avere	Azure Files
Global Content Delivery	CloudFront	Cloud CDN	Delivery Network
Managed Data Warehouse	Redshift	Big Query	SQL Warehouse

Cloud Comparison

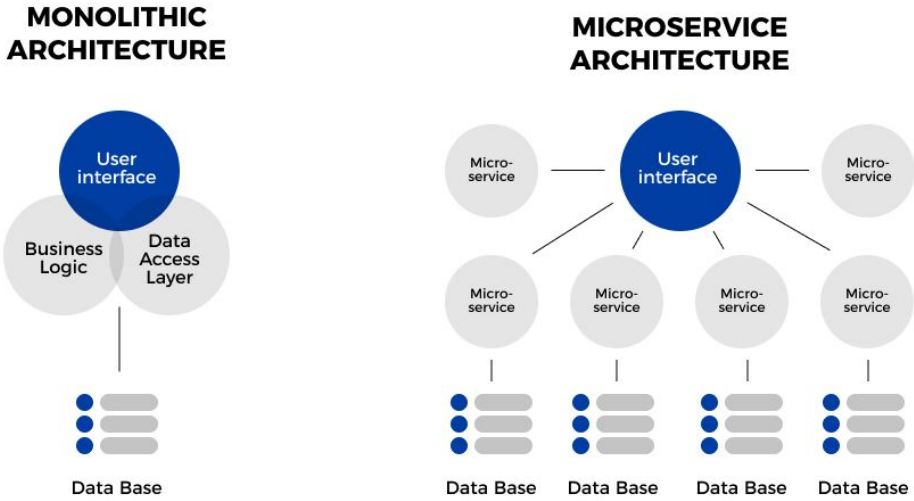
Azure vs. AWS vs. Google Compute

	Azure	aws	
Available Regions	Azure Regions	AWS Regions and Zones	Google Compute Regions & Zones
Compute Services	Virtual Machines	Elastic Compute Cloud (EC2)	Compute Engine
App Hosting	Azure Cloud Services	Amazon Elastic Beanstalk	Google App Engine
Serverless Computing	Azure Functions	AWS Lambda	Google Cloud Functions
Container Support	Azure Container Service	EC2 Container Service	Container Engine
Scaling Options	Azure Autoscale	Auto Scaling	Autoscaler
Object Storage	Azure Blob Storage	Amazon Simple Storage (S3)	Cloud Storage
Block Storage	Azure Managed Storage	Amazon Elastic Block Storage	Persistent Disk
Content Delivery Network (CDN)	Azure CDN	Amazon CloudFront	Cloud CDN
SQL Database Options	Azure SQL Database	Amazon RDS	Cloud SQL
NoSQL Database Options	Azure DocumentDB	AWS DynamoDB	Cloud Datastore
Virtual Network	Azure Virtual Network	Amazon VPC	Cloud Virtual Network
Private Connectivity	Azure Express Route	AWS Direct Connect	Cloud Interconnect
DNS Services	Azure Traffic Manager	Amazon Route 53	Cloud DNS
Log Monitoring	Azure Operational Insights	Amazon CloudTrail	Cloud Logging
Performance Monitoring	Azure Application Insights	Amazon CloudWatch	Stackdriver Monitoring
Administration and Security	Azure Active Directory	AWS Identity and Access Management (IAM)	Cloud Identity and Access Management (IAM)
Compliance	Azure Trust Center	AWS CloudHSM	Google Cloud Platform Security
Analytics	Azure Stream Analytics	Amazon Kinesis	Cloud Dataflow
Automation	Azure Automation	AWS Opsworks	Compute Engine Management
Management Services & Options	Azure Resource Manager	Amazon Cloudformation	Cloud Deployment Manager
Notifications	Azure Notifcation Hub	Amazon Simple Notification Service (SNS)	None
Load Balancing	Load Balancing for Azure	Elastic Load Balancing	Cloud Load Balancing

Microservice Architecture vs Monolith

Monolith vs Microservices

- Single unit vs distributed



Microservice Design Principles

- Single responsibility
- Independent deployment
- Explicit interfaces
- Decentralized data

Single Responsibility

What it means:

- Each microservice owns one business capability
- Changes in one domain should not force changes in others

Why it matters:

- Smaller codebases ⟶ easier reviews
- Smaller blast radius during failures or incidents

Security / audit perspective:

- Clear ownership of:
    - data
    - logic
    - access rights
- Easier threat modeling and data classification

Independent Deployment

What it means

- Each service:
  - has its own CI/CD pipeline
  - can be deployed independently
  - has its own lifecycle

Why it matters

- No “big bang” deployments
- Faster incident response and rollback

Independent Deployment

Security / audit perspective

- Every deployment:
  - is logged
  - has an author
  - has a timestamp
  - Strong change-management evidence

Trade-offs

- More pipelines to govern
- Higher operational complexity

Explicit Interfaces

What it means

- Services communicate only via:
  - APIs
  - events
  - No shared databases or memory

Why it matters

- Clear contracts
- Easier versioning and compatibility control

Explicit Interfaces

Security / audit perspective

- Natural enforcement points:
  - authentication
  - authorization
  - rate limiting
- Better observability and access logging

Red flag

- One service reading another service’s database

Decentralized Data

What it means

- Each service owns:
  - its data
  - its schema
  - its storage technology

Why it matters

- Loose coupling
- Independent scaling and evolution

Security / audit perspective

- Stronger data isolation
- Different data classifications per service
- Reduced impact of a data breach

What Is the 12-Factor App?

- Not a framework
- Not a tool
- Design methodology
- Cloud-native foundation

Twelve-Factor App Principles

1	Codebase	Application resources should have a version-managed, source-code repository
2	Dependencies	Should be set up in app manifest or configuration file and managed externally
3	Config	Configurations of an app need to be stored independently as environment variables
4	Backing Services	Services like databases, messaging systems, SMTP services, etc. should be architected as external resources
5	Build, Release, Run	There should be three independent steps of a deployment process
6	Stateless Processes	Apps should have the provision to be served by multiple stateless, independent processes
7	Port Binding	Apps directly bind to a port and responds to incoming requests
8	Concurrency	Apps need to be broken down into multiple modules for scaling
9	Disposability	Apps should maximize robustness with quick startup and easy shutdown
10	DEV/PROD Parity	Development, staging, and production should be kept as similar as possible
11	Logs	Treat logs as event streams
12	Admin Processes	Admin tasks should be run as one-off processes

Rheinwerk Computing

12-Factor: Config

- No secrets in code
- Runtime injection
- Central secret management

12-Factor: Build, Release, Run

- Build = immutable artifact
- Release = artifact + config
- Run = execution only

12-Factor: Stateless & Logs

- Stateless services
- Logs as event streams
- Central logging

12-Factor vs Monolith (Audit)

- Better evidence
- Automation

12-Factor vs Monolith (Audit)

Area	Monolith	12-Factor App
Config	Files	Env / Secret Manager
Logs	Local	Central
Deploy	Manual	Automated
Evidence	Weak	Strong

Microservice Patterns

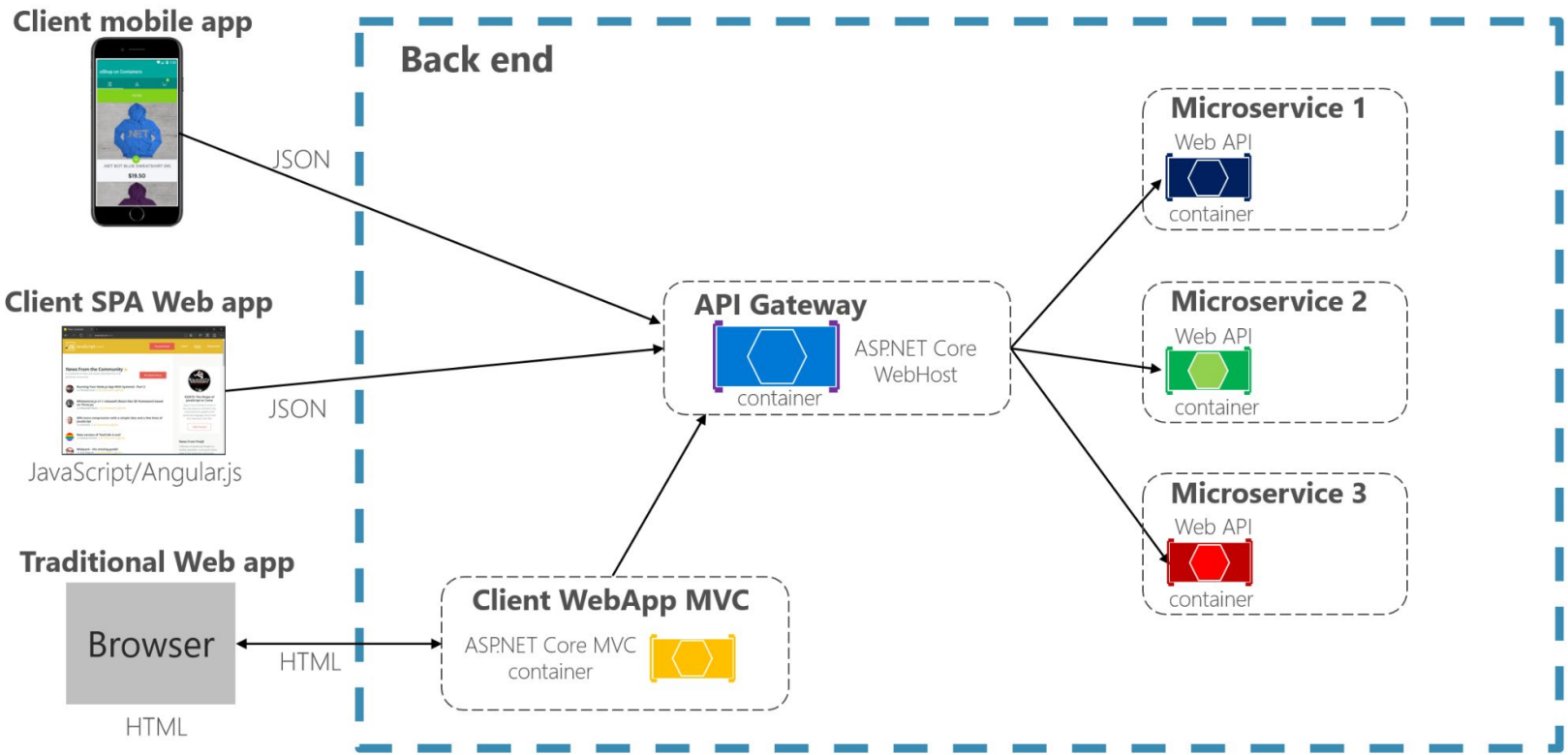
- Entry & Communication Patterns
  - API Gateway
  - Backend for Frontend
- Reliability & Resilience Patterns
  - Circuit Breaker
  - Retry with Backoff
  - Bulkhead
- Data Management Patterns
  - Database per Service
  - Saga
- Observability & Operations Patterns
  - Centralized Logging
  - Health Check
- Security Patterns
  - Zero Trust / mTLS (Service Mesh)

More

API Gateway

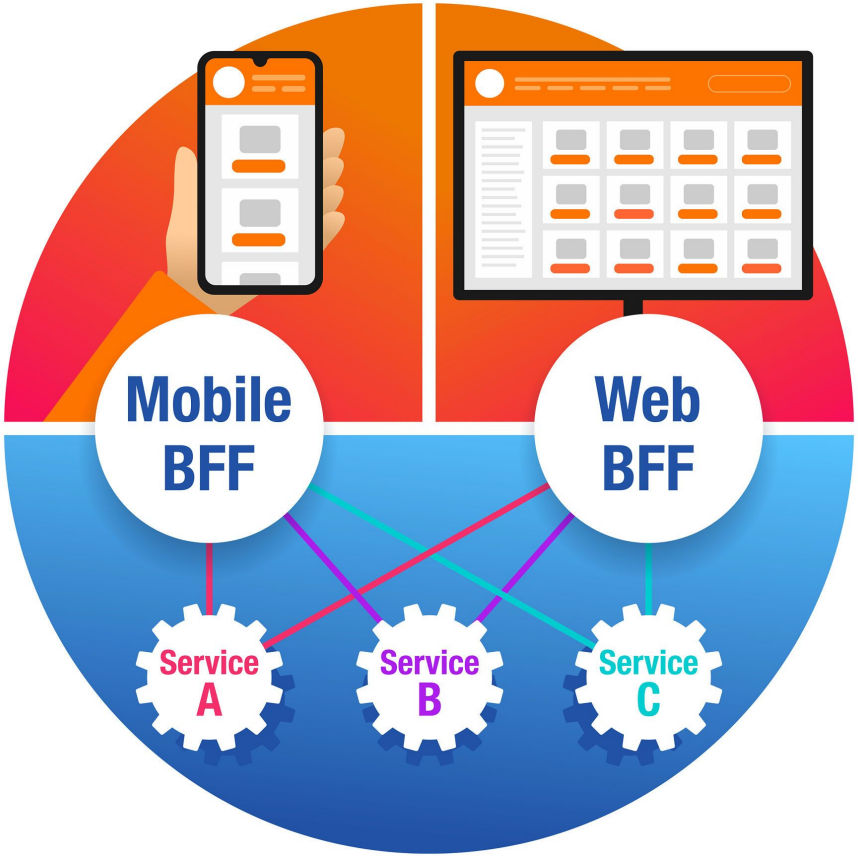
- What it is
  - Single entry point for external traffic to microservices.
- Pros
  - Centralized authentication & authorization
  - Rate limiting, request validation
  - Good audit control point
- Cons
  - Can become a bottleneck
  - High-value attack target
- When to use
  - Always for external access

Using a single custom **API Gateway service**



Backend for Frontend (BFF)

- What it is
  - Dedicated gateway per client type (web, mobile, partner API).
- Pros
  - Reduced attack surface per client
  - Clear separation of access rules
- Cons
  - More services to maintain
- When to use
  - Multiple clients with different needs


Mobile
BFF
Web
BFF
Service
A
Service
B
Service
C

Circuit Breaker

What it is

- Stops requests to failing services.

Pros

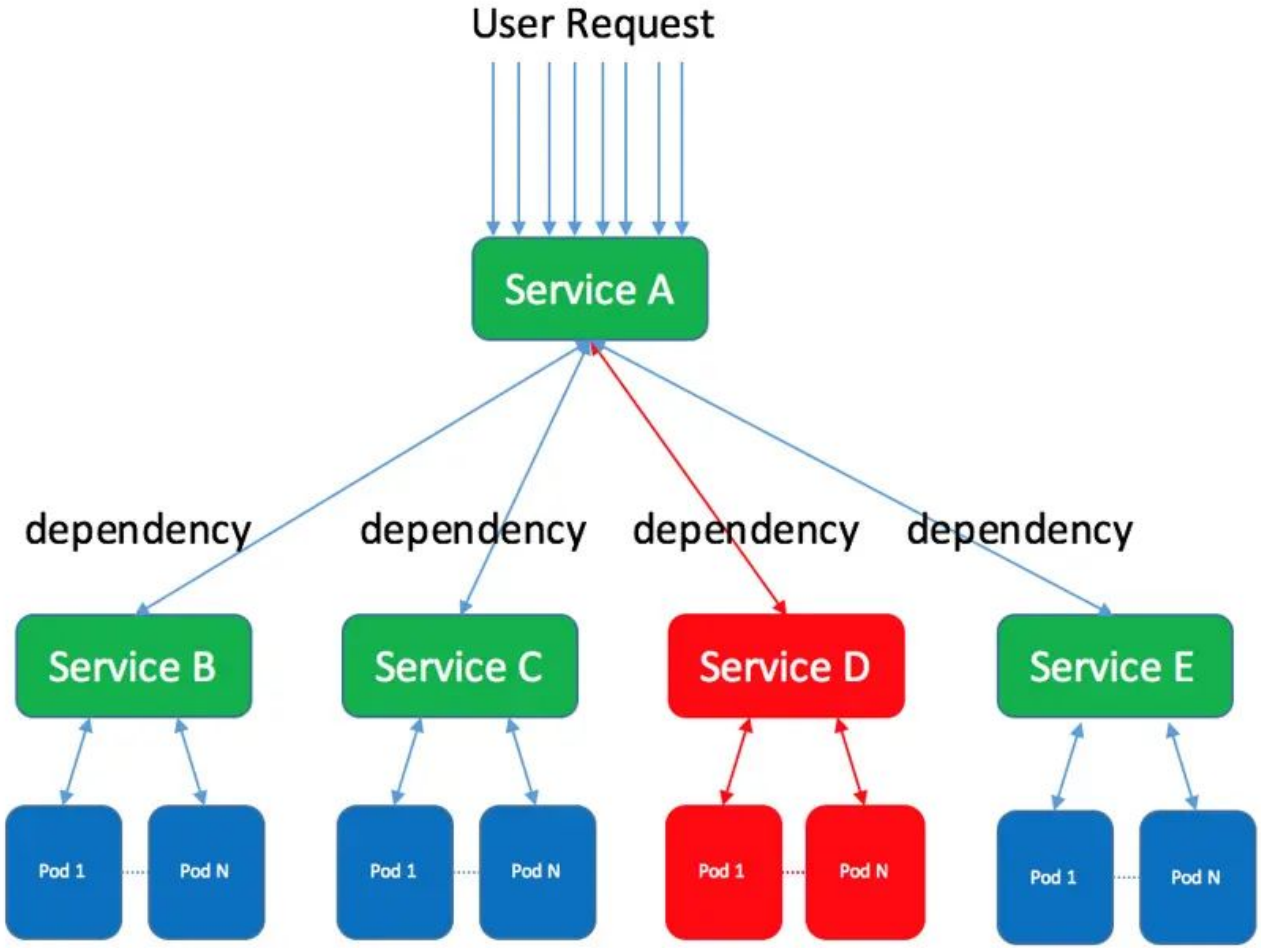
- Prevents cascading failures
- Improves system stability

Cons

- Requires careful tuning

When to use

- Systems with many service dependencies


User Request
Service A
dependency
dependency
dependency
dependency
Service B
Service C
Service D
Service E
Pod 1
Pod N
Pod 1
Pod N
Pod 1
Pod N
Pod 1
Pod N

Retry with Backoff

What it is

- Retries failed requests with increasing delays.

Pros

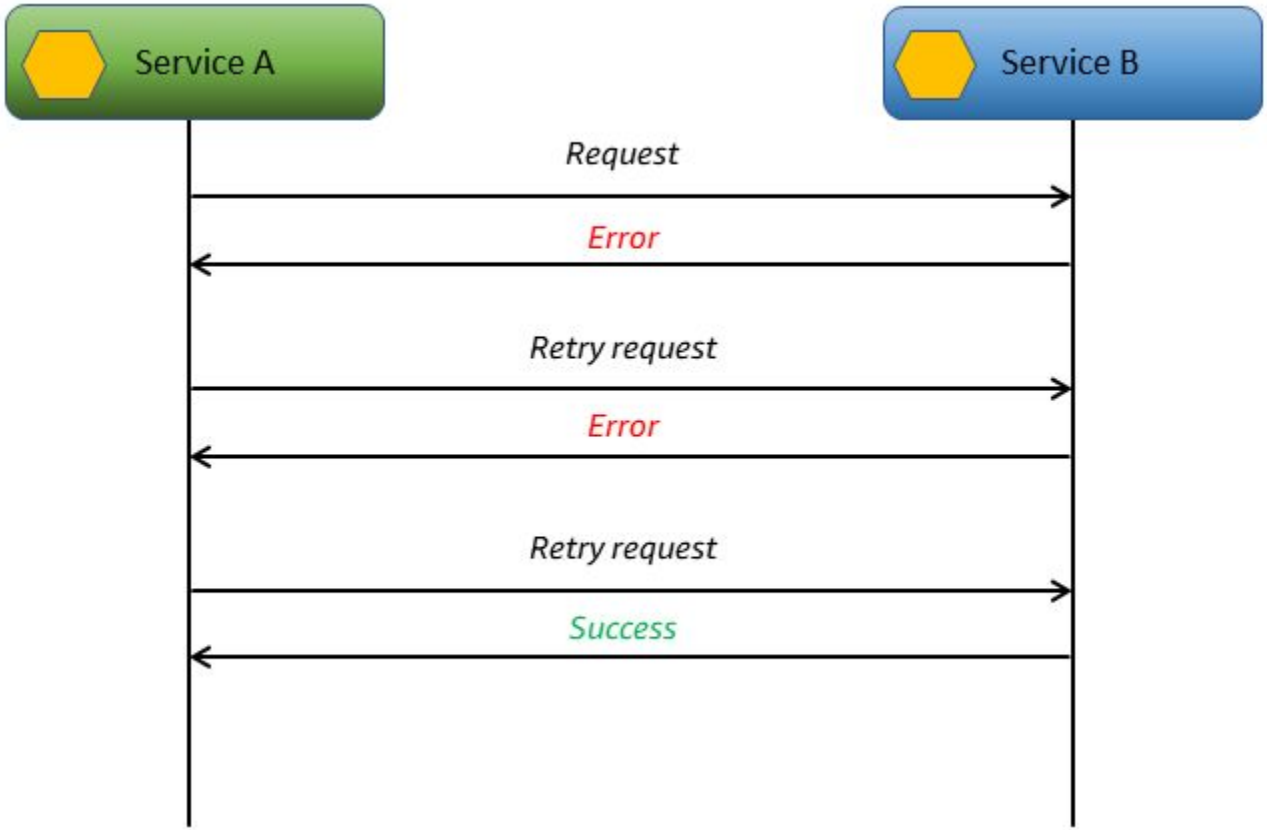
- Handles transient failures

Cons

- Can amplify load if misused

When to use

- Idempotent operations only


Service A
Service B
Request
Error
Retry request
Error
Retry request
Success

Bulkhead

What it is

- Resource isolation per service or dependency.

Pros

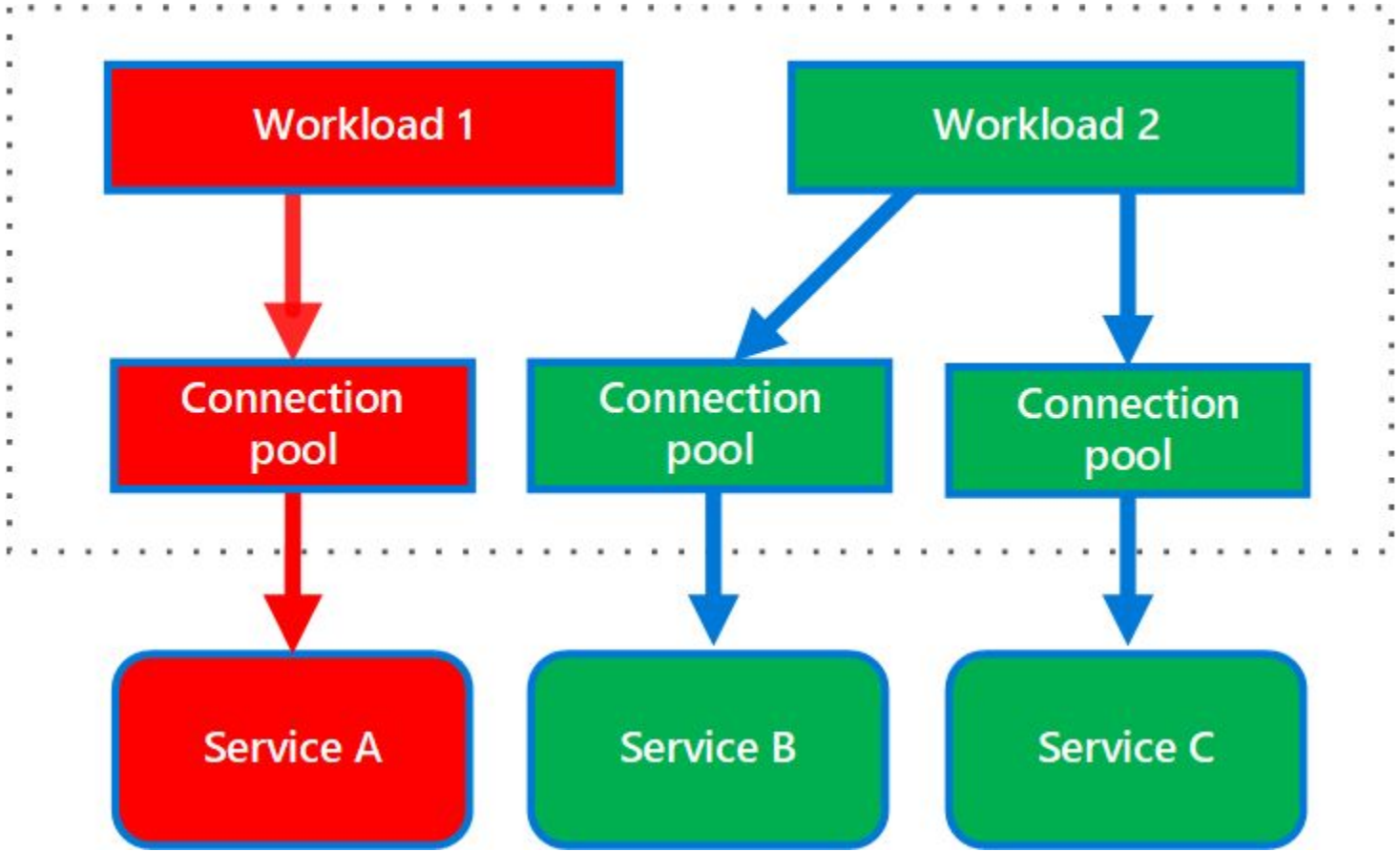
- Limits blast radius
- Improves fault isolation

Cons

- Resource fragmentation

When to use

- Critical or unstable dependencies


Workload 1
Workload 2
Connection pool
Connection pool
Connection pool
Service A
Service B
Service C

Database per Service

What it is

- Each service owns its database.

Pros

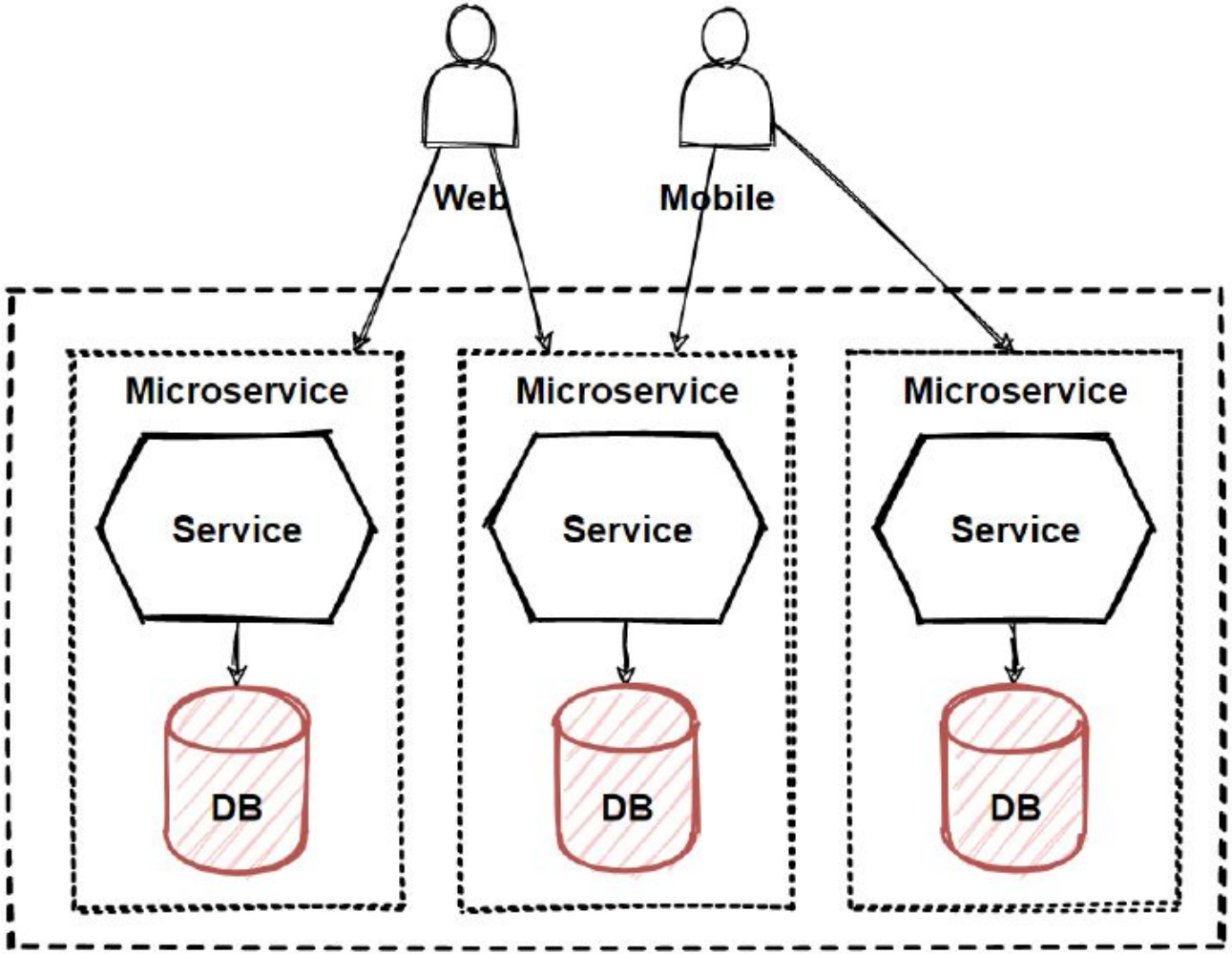
- Strong data isolation
- Independent scaling

Cons

- Harder cross-service queries

When to use

- Core microservice architectures


Web
Mobile
Microservice
Service
DB
Microservice
Service
DB
Microservice
Service
DB

Saga

What it is

- Manages distributed transactions using events or orchestration.

Pros

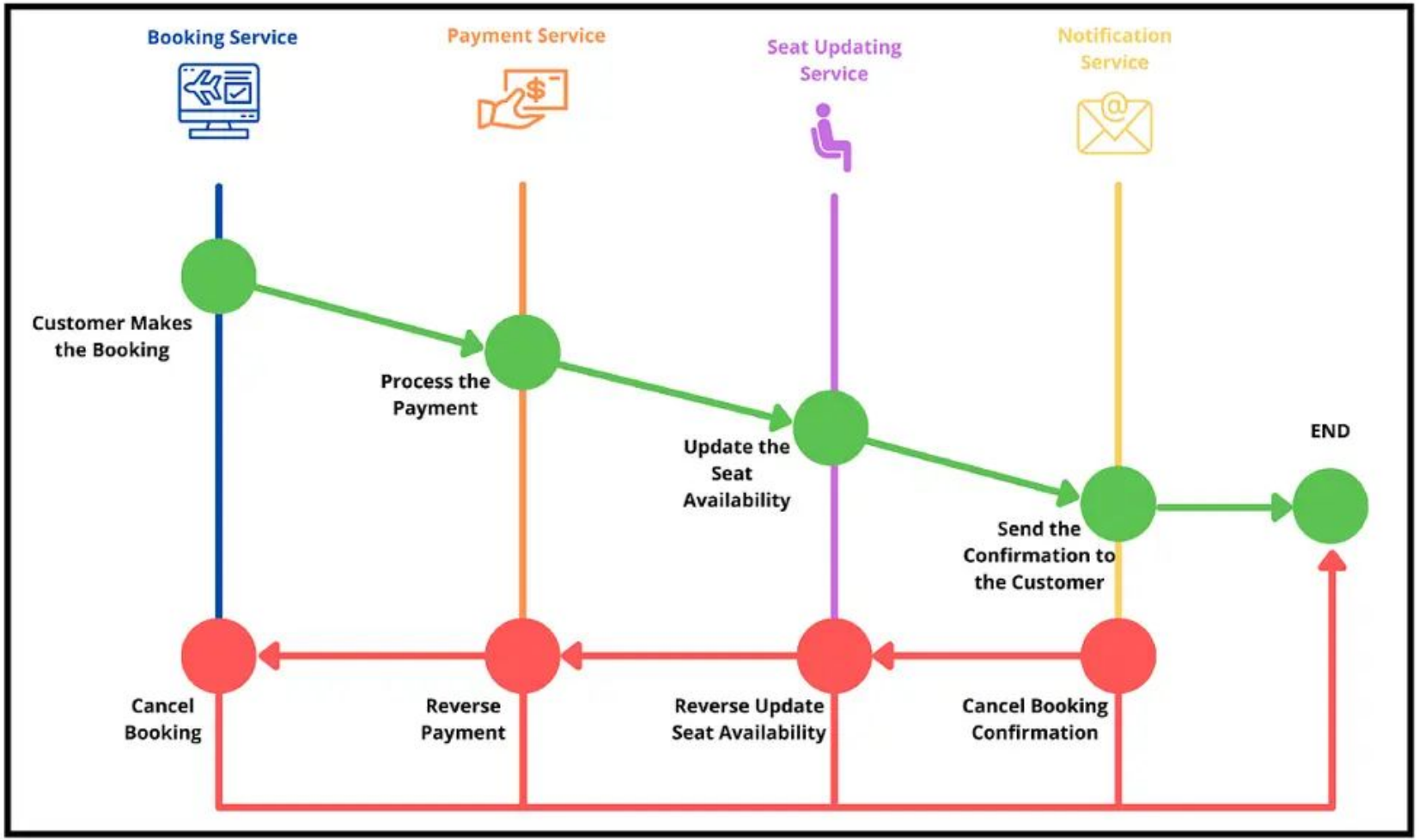
- No global locks
- Clear business workflows

Cons

- Complex error handling

When to use

- Multi-step business transactions


Booking Service
Payment Service
Seat Updating Service
Notification Service
Customer Makes the Booking
Process the Payment
Update the Seat Availability
Send the Confirmation to the Customer
END
Cancel Booking
Reverse Payment
Reverse Update Seat Availability
Cancel Booking Confirmation

Centralized Logging

What it is

- Logs streamed to a central system.

Pros

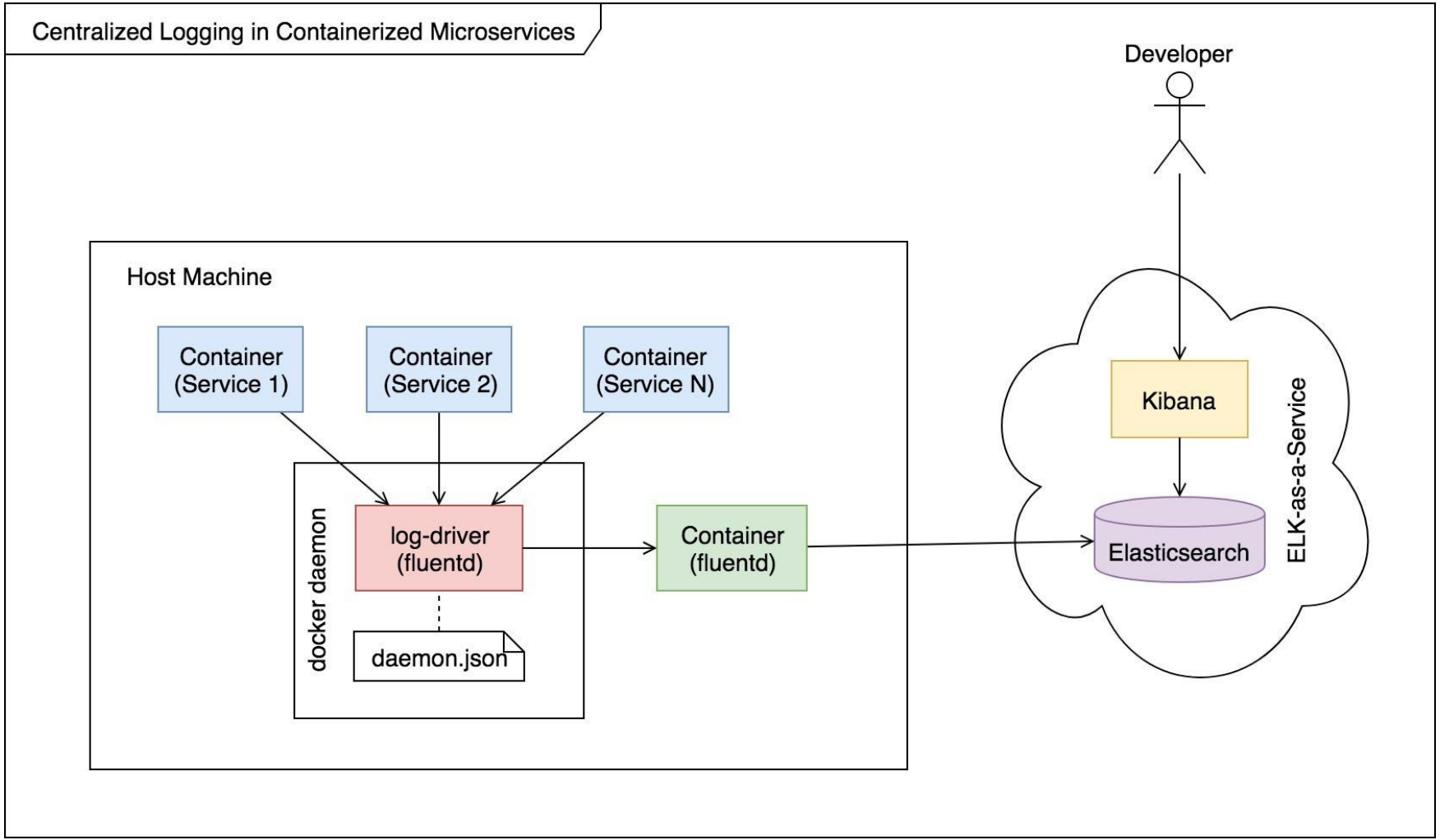
- Forensics & compliance
- Strong audit evidence

Cons

- Storage & retention cost

When to use

- Always (mandatory in regulated environments)


Centralized Logging in Containerized Microservices
Developer
Host Machine
Container (Service 1)
Container (Service 2)
Container (Service N)
docker daemon
log-driver (fluentd)
daemon.json
Container (fluentd)
Kibana
Elasticsearch
ELK-as-a-Service

Health Check (Liveness / Readiness)

What it is

- Endpoints exposing service health.

Pros

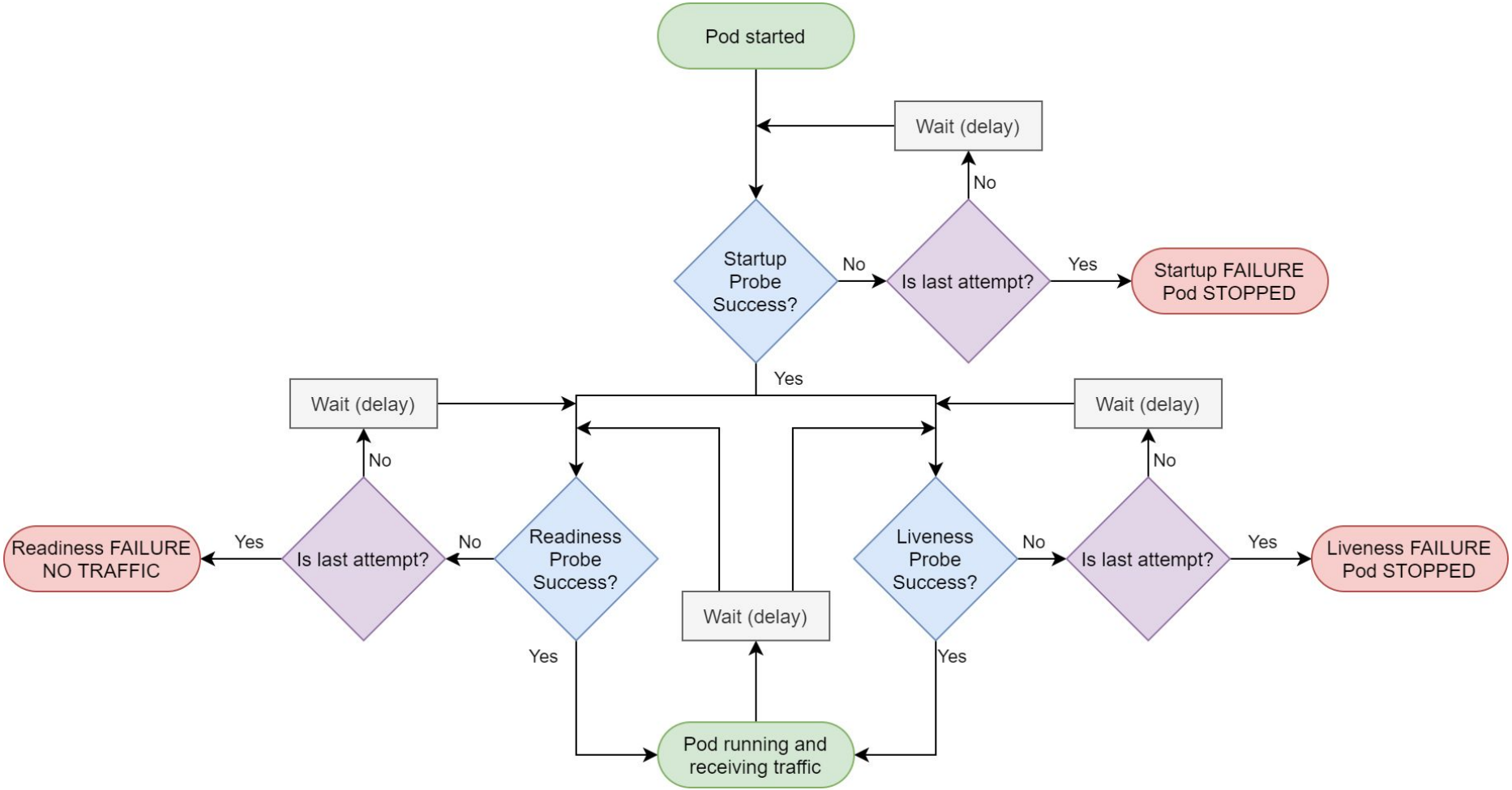
- Enables auto-healing
- Safe rollouts

Cons

- Poor checks give false confidence

When to use

- Every production service


Pod started
Wait (delay)
No
Startup Probe Success?
No
Is last attempt?
Yes
Startup FAILURE Pod STOPPED
Yes
Wait (delay)
Wait (delay)
No
No
Readiness FAILURE NO TRAFFIC
Yes
Is last attempt?
No
Readiness Probe Success?
Liveness Probe Success?
No
Is last attempt?
Yes
Liveness FAILURE Pod STOPPED
Wait (delay)
Yes
Yes
Pod running and receiving traffic

Service Mesh (mTLS / Zero Trust)

What it is

- Mutual TLS and identity-based communication between services.

Pros

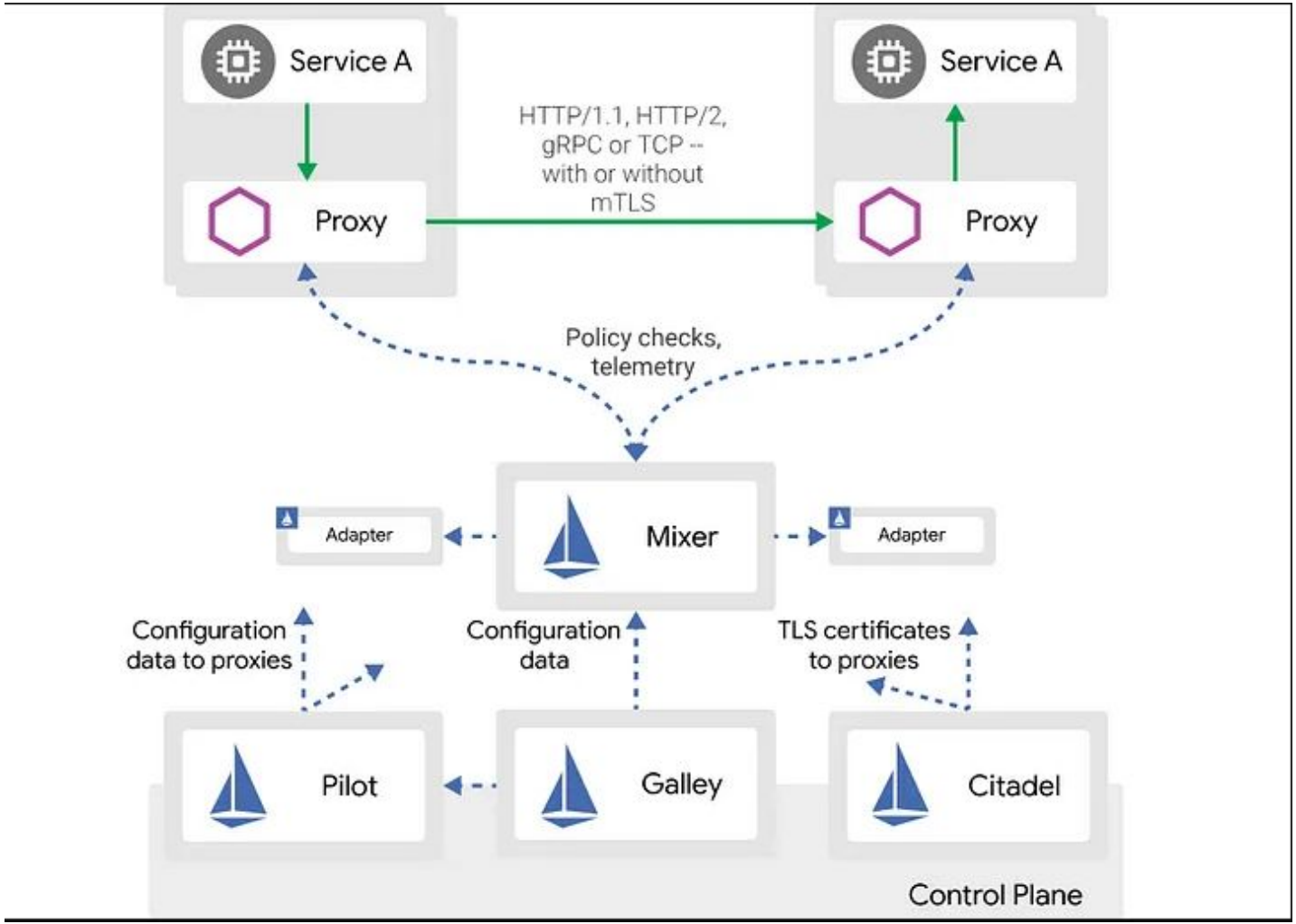
- Encrypted traffic
- Strong service identity
- Fine-grained authorization

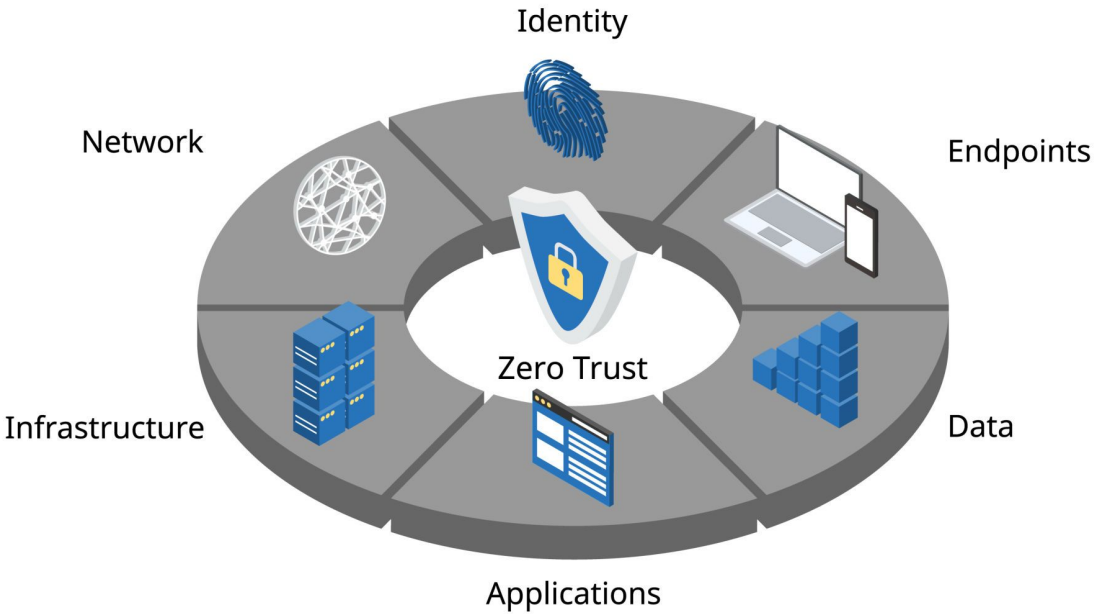
Cons

- Operational complexity
- Performance overhead

When to use

- Sensitive data
- Regulated environments

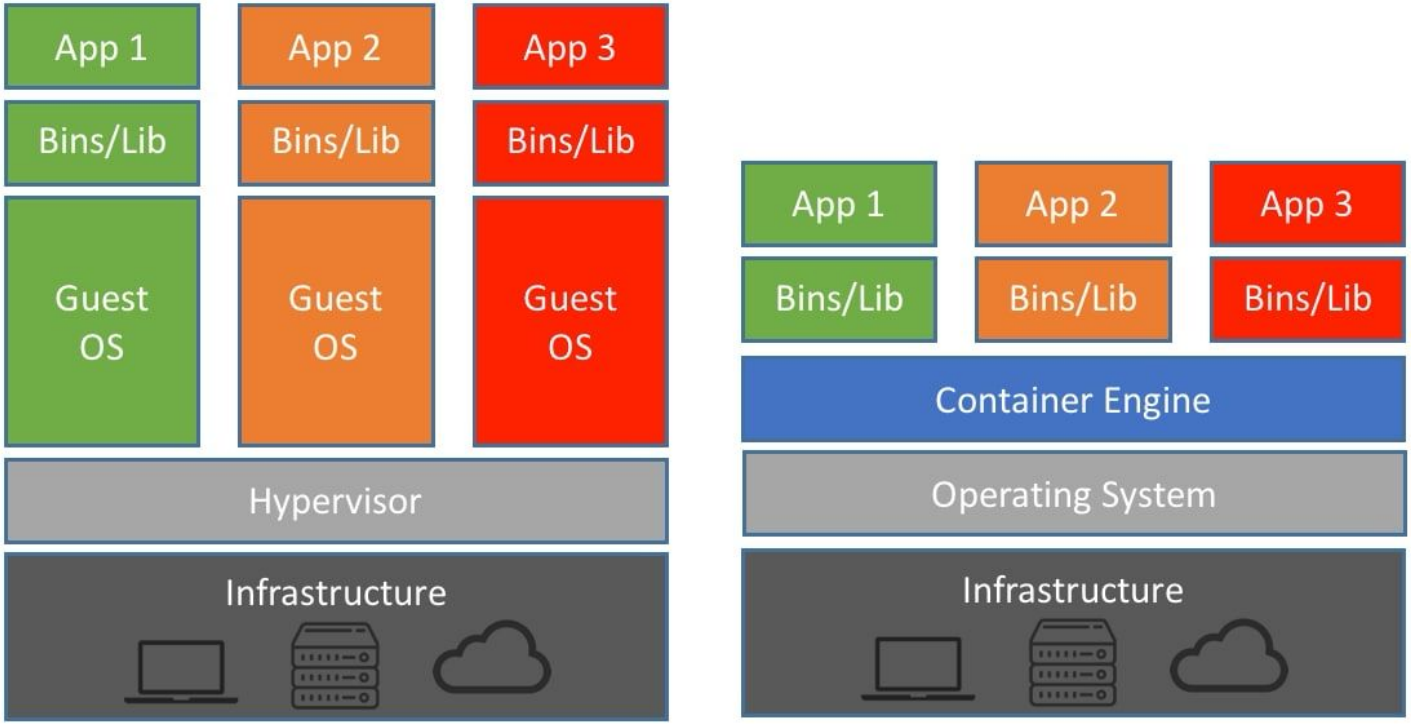

Service A
Service A
HTTP/1.1, HTTP/2,
gRPC or TCP --
with or without
mTLS
Proxy
Proxy
Policy checks,
telemetry
Adapter
Mixer
Adapter
Configuration
data to proxies
Configuration
data
TLS certificates
to proxies
Pilot
Galley
Citadel
Control Plane


Identity
Network
Endpoints
Zero Trust
Infrastructure
Data
Applications

Containerization and Orchestration (Docker, Kubernetes)

What Are Containers?

- Containers are lightweight, isolated environments used to run applications
- They package:
  - application code
  - runtime
  - libraries and dependencies
- Containers share the host operating system kernel
- They start quickly and use fewer resources than virtual machines
- Containers are immutable at runtime (rebuilt, not modified)
- They are designed to be portable and consistent across environments



Machine Virtualization

Containers

What Are Container Images?

- A container image is an immutable, versioned artifact used to run containers
- It contains:
  - application code
  - runtime
  - libraries and dependencies
- Images are typically stored in container registries
- An image represents a deployment unit and a supply-chain artifact
- Images are identified by:
  - tags (e.g. v1.2.3)
  - digests (SHA256 – preferred for security)

What Are Container Images?

Security relevance

- Images define what is actually running in production
- Vulnerabilities in images propagate to all running containers
- Image immutability enables rollback and auditability

What Is a Dockerfile?

- A container image is an immutable, versioned artifact used to run containers
- It contains:
  - application code
  - runtime
  - libraries and dependencies
- Images are typically stored in container registries
- An image represents a deployment unit and a supply-chain artifact
- Images are identified by:
  - tags (e.g. v1.2.3)
  - digests (SHA256 – preferred for security)

What Is a Dockerfile?

Security relevance

- Images define what is actually running in production
- Vulnerabilities in images propagate to all running containers
- Image immutability enables rollback and auditability

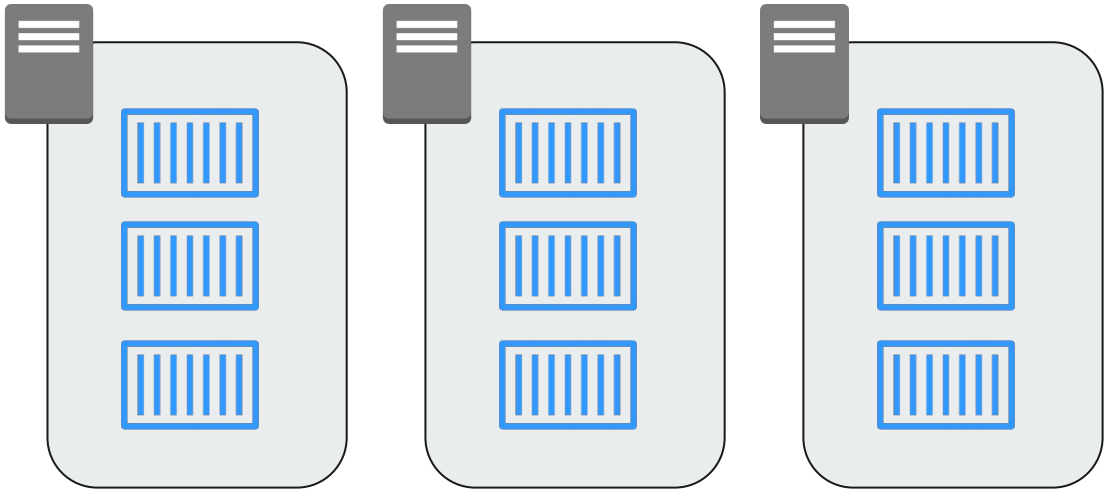
Example Dockerfile

```
# Base image
FROM python:3.12-slim
# Working directory inside the container
WORKDIR /app
# Copy dependencies first (for caching)
COPY requirements.txt .
# Install dependencies
RUN pip install --no-cache-dir -r requirements.txt
# Copy the application code
COPY . .
# Expose application port
EXPOSE 8080
# Define startup command
CMD ["python", "app.py"]
```

What is Kubernetes?

“Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications.” – CNCF

The Problem Before Kubernetes

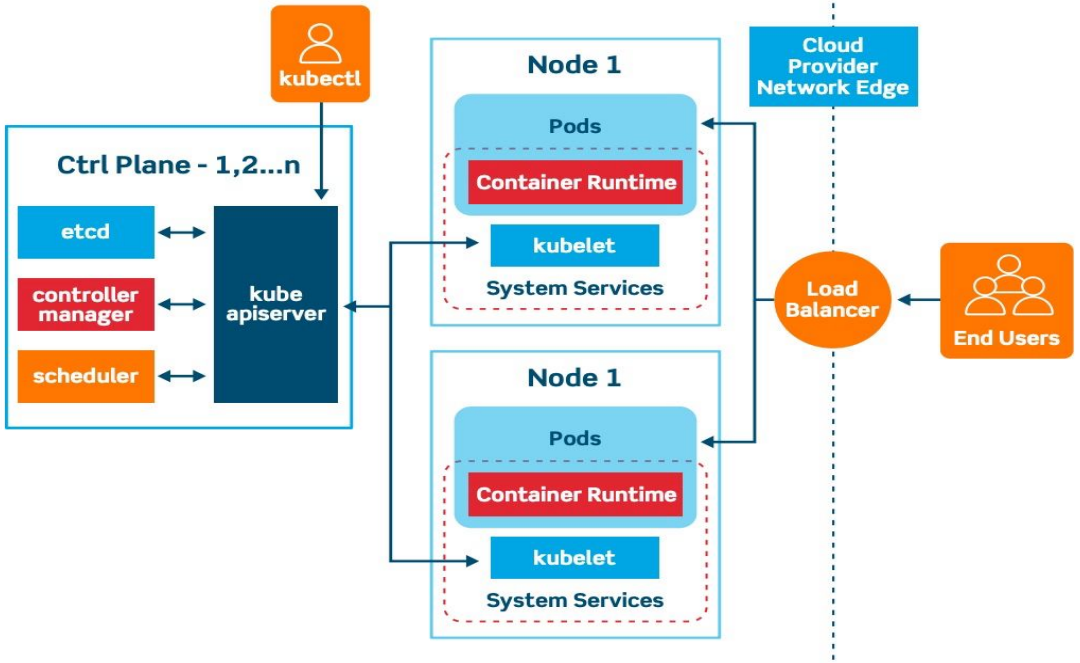


- Manual container management and updates
- Difficult scaling and recovery
- Updates can caused downtime
- Managing multiple containers was becoming difficult and time-consuming.
- No centralized control or observability

Core Terminology

Term	Description	Example
Cluster	A set of machines managed by Kubernetes	Control + Worker nodes
Node	A single machine (virtual or physical)	Worker node
Pod	The smallest deployable unit	One or more containers
Service	Stable access point to Pods	ClusterIP, NodePort
Deployment	Controller that manages Pods and configuration	kubectl apply -f
Namespace	Logical resource grouping	dev, prod, staging

Kubernetes Architecture



API Server

- Acts as the central communication hub for the entire cluster.
- Exposes the Kubernetes REST API, which all components and clients (kubectl, controllers) use.
- Responsible for:
  - Validating and authenticating requests.
  - Persisting data in etcd.
  - Serving as the single source of truth for cluster state.

etcd

- A distributed key–value database storing the full cluster configuration and state.
- Keeps both desired and current state of all objects.
- Critical component – regular backups are mandatory.
- Used by API Server for every read/write operation.

Controller Manager

- Runs multiple internal controllers that constantly reconcile desired vs. actual state.
- Examples:
  - Node Controller – monitors node health.
  - ReplicaSet Controller – ensures correct number of Pod replicas.
  - Deployment Controller – manages rolling updates and rollbacks.
- If a Pod or Node disappears, the Controller Manager takes corrective action.

Scheduler

- Decides which node should host a newly created Pod.
- Evaluates:
  - Resource availability (CPU, memory).
  - Node taints and tolerations.
  - Pod affinity and anti-affinity rules.
- Once a node is chosen, the Scheduler communicates the decision to the API Server.

Cloud Controller Manager

- Integrates Kubernetes with external cloud providers (e.g., AWS, GCP, Azure).
- Manages:
  - Node lifecycle events (add/remove).
  - Load balancer provisioning.
  - Persistent volume integration.
- Makes Kubernetes cloud-aware while keeping Control Plane logic provider-agnostic.

Kubelet

- Primary node agent that communicates directly with the API Server.
- Ensures all Pods assigned to the node are running and healthy.
- Core responsibilities:
  - Registers the node with the cluster.
  - Monitors Pod specs and reports status back to the Control Plane.
  - Restarts failed containers according to Pod definitions.
- Also handles liveness and readiness probes for health checking.

kube-proxy

- Maintains the network connectivity rules inside the node.
- Implements Service load balancing and Pod-to-Pod routing.
- Works in one of two modes:
  - iptables mode – creates NAT rules for routing.
  - IPVS mode – high-performance connection-based load balancing.
  - nftables – modern replacement for iptables using the Linux kernel's netfilter framework;
  - provides better performance, rule grouping, and scalability on new kernels.
- Ensures that traffic to a Service is evenly distributed among Pods.

Container Runtime

- Responsible for running and managing containers defined in Pods.
- Implements the Container Runtime Interface (CRI).
- Common implementations:
  - containerd – lightweight, high-performance runtime (default on most distros).
  - CRI-O – optimized for Kubernetes, used by Red Hat and OpenShift.
  - (Docker Engine deprecated since K8s v1.24)
- Handles:
  - Image pulling, container lifecycle.
  - Resource isolation via cgroups and namespaces.

Pod: The Smallest Deployable Unit

- The smallest deployable and schedulable unit in Kubernetes.
- Encapsulates one or more tightly coupled containers.
- Containers in the same Pod:
  - Share the same network namespace (same IP).
  - Can communicate via localhost.
  - Optionally share storage volumes.

Init Containers

- Special containers that run before regular application containers.
- Execute one-time setup tasks, such as:
  - Waiting for a dependency (DB, API).
  - Initializing configuration or data.
  - Applying schema migrations.
- Each init container must complete successfully before the next one (or main containers) start.

Sidecar

- A secondary container inside the same Pod as the main application.
- Adds supporting functionality without modifying the main app code.

Adapter

- A secondary container inside the same Pod that transforms or normalizes data.
- Used when the main application outputs logs, metrics, or data in a non-standard format.
- The adapter translates it into a format that the rest of the platform understands.

Ambasador

- A container in the same Pod that acts as a proxy or gateway for outbound connections.
- The main container connects to the ambassador over localhost.
- The ambassador handles communication with remote services — databases, APIs, or external networks.

Deployment

- A Deployment is a Kubernetes object used to manage stateless applications
- It defines the desired state of Pods (how many, which image, how they are configured)
- Kubernetes automatically ensures the actual state matches the desired state
- A Deployment manages ReplicaSets, which in turn create and manage Pods
- It supports rolling updates and rollbacks
- It enables scaling (manual or automatic via HPA)
- It provides self-healing by recreating failed Pods
- Changes to a Deployment are versioned and auditable

ConfigMap

- A Kubernetes object for storing non-sensitive configuration data.
- Key-value pairs such as:
    - feature flags,
    - log levels,
    - service URLs,
    - runtime tuning parameters.
- Lets you change behavior of an app without rebuilding the container image.

Secret

- A Kubernetes object that stores small amounts of sensitive data - such as passwords, tokens, SSH keys, or certificates.
- Designed to separate confidential information from container images and configuration.
- Mounted or injected at runtime (never baked into images).

Service: Stable Network Identity

- Provides a stable access endpoint for a dynamic set of Pods.
- Abstracts Pod IP changes using a virtual IP and DNS name.
- Implements load balancing across replicas.
- Common types:
  - ClusterIP – internal cluster access (default).
  - NodePort – exposes on node ports (30000–32767).
  - LoadBalancer – integrates with external cloud LB.

Job

- A controller that runs Pods until they successfully complete.
- Guarantees that a defined number of Pods finish successfully.
- Retries automatically on failure (depending on configuration).
- Used for batch processing, migrations, and one-time tasks.

CronJob

- Creates Jobs on a time-based schedule, like Linux cron.
- Each schedule execution spawns a new Job.
- Great for:
  - Backups
  - Reports
  - Database maintenance
  - Cleanup tasks

Ingress

- Ingress is a Kubernetes object that defines HTTP/HTTPS routing rules
- It exposes services outside the cluster using:
  - hostnames
  - paths
- Ingress works at Layer 7 (HTTP)
- It supports:
  - TLS termination
  - host-based routing
  - path-based routing
- Ingress is only a configuration object, not a traffic handler

What Is an Ingress Controller?

- An Ingress Controller is the component that:
  - watches Ingress resources
  - implements the routing rules
- It is a running application inside the cluster
- Common examples:
  - NGINX Ingress Controller
  - HAProxy
  - Cloud provider load balancer controllers
- Responsible for:
  - traffic forwarding
  - TLS handling
  - applying security features (rate limiting, headers)

What Is an Ingress Controller?

Security / audit relevance

- Central enforcement point for:
  - TLS
  - authentication
  - request logging
- Misconfiguration can expose the entire cluster

Gateway API

- Gateway API is the next-generation Kubernetes API for traffic management
- Designed to replace and extend Ingress
- More expressive and more structured
- Built around clear roles and responsibilities
- Key design goals
  - Standardized
  - Extensible
  - Role-oriented (platform vs application teams)

Probes, Autoscaling, Rollouts

- Liveness vs readiness
- HPA, VPA
- Canary / Blue-Green

What are probes?

- Probes are health checks executed by Kubernetes against containers
- They determine whether a container is healthy and ready to receive traffic

Liveness Probe

What it does

- Checks if the application is still running correctly
- If it fails repeatedly → container is restarted

Why it matters

- Detects deadlocks or stuck processes
- Enables self-healing

Security / audit perspective

- Prevents long-running faulty or compromised processes
- Provides evidence of automated remediation

Common mistake

- Using liveness checks for dependency health (DB, API)

Readiness Probe

What it does

- Checks if the application is ready to accept traffic
- If it fails → Pod is removed from Service endpoints
- Container is not restarted

Why it matters

- Zero-downtime deployments
- Safe startup and graceful degradation

Security / audit perspective

- Reduces exposure to half-initialized or misconfigured services
- Limits blast radius during incidents

What is autoscaling?

- Autoscaling dynamically adjusts the number of Pods
- Based on:
  - CPU
  - memory
  - custom or external metrics

Horizontal Pod Autoscaler (HPA)

What it does

- Scales Pods up or down automatically
- Reacts to load in near real time

Why it matters

- Handles traffic spikes without manual intervention
- Improves availability and resilience

Security implications

- Reduces DoS impact by absorbing traffic
- Prevents single-instance overload
- Requires quotas and limits to avoid cost abuse

Audit evidence

- HPA configuration
- Metrics history
- Scaling events (who, when, why)

Vertical Pod Autoscaler (VPA)

- Vertical Pod Autoscaler (VPA) automatically adjusts:
  - CPU requests
  - memory requests
  - It optimizes resource allocation per Pod
  - VPA focuses on right-sizing, not scaling out
- Works best for:
  - stable workloads
  - batch jobs
  - services with predictable traffic

What is a rollout?

- A rollout is the process of deploying a new application version
- Managed automatically by Kubernetes Deployments
- By default Kubernetes only support RollingUpdate and Replace other like Canary or A/B need to be handle externally

Rolling Update (default)

What it does

- Gradually replaces old Pods with new ones
- Keeps the application available during updates

Why it matters

- No downtime
- Reduced deployment risk
- Security / audit perspective

Each rollout:

- is versioned
- is traceable
- can be rolled back
- Strong change-management evidence

What “Security” Mean in Kubernetes?

Kubernetes security covers multiple layers, not just user access.

It involves securing:

- The API Server — who can talk to the control plane.
- Workloads — what containers and Pods can do.
- The Network — which Pods can talk to each other.
- The Supply Chain — ensuring trusted images and manifests.

Kubernetes Security Responsibilities

Developers:

- Avoid running containers as root
- Limit container privileges
- Store secrets safely

Cluster Operators:

- Manage authentication & RBAC
- Apply PodSecurity standards
- Enforce admission policies

Infrastructure / Platform Team:

- Secure API server and etcd
- Patch the control plane
- Configure audit logs and encryption

ServiceAccount

A ServiceAccount (SA) is an identity used by processes running inside the cluster to authenticate to the Kubernetes API.

- Every Pod runs under a ServiceAccount context.
- ServiceAccounts are meant for machines, Pods, or controllers — not human users.
- They’re essential for fine-grained access control via RBAC.

Continuous Integration & Continuous Delivery/Deployment

What is CI/CD

Continuous Integration (CI)

What it means

- Developers frequently merge code into a shared repository
- Each change automatically triggers:
- build
- tests
- static analysis

Why it matters

- Early detection of bugs and misconfigurations
- Prevents broken or insecure code from progressing further

Continuous Integration (CI)

Security / audit perspective

- CI enforces shift-left security
- Security checks become:
- repeatable
- mandatory
- logged
- Eliminates “it worked on my machine”

Audit evidence

- Pipeline execution logs
- Test results
- Static analysis reports
- Commit -> pipeline traceability

Continuous Delivery/Deployment (CD)

What it means

- Continuous Delivery: code is always deployable, release is manual
- Continuous Deployment: code is deployed automatically

Why it matters

- Fast and controlled releases
- Reduced human error during deployments

Continuous Delivery/Deployment (CD)

Security / audit perspective

- Deployment logic is code, not a human procedure
- Clear separation:
- who can change code
- who can approve release
- Rollbacks are automated and logged

Audit evidence

- Deployment history
- Approval records
- Environment-specific policies
- Git tags / release metadata

CI/CD Tools Landscape

Source Control

- Git repositories are the single source of truth
- Every change is:
  - versioned
  - attributed to an identity
  - reviewable

Audit relevance

- Strong non-repudiation
- Full change history

CI Systems (Build & Test)

Examples:

- GitHub Actions
- GitLab CI
- Jenkins

What they do

- Build artifacts
- Run tests
- Enforce quality and security checks

Audit relevance

- Evidence of:
    - automated validation
    - policy enforcement
    - failed vs successful runs

CD / GitOps Tools (Deploy)

Examples:

- ArgoCD
- Flux

What they do

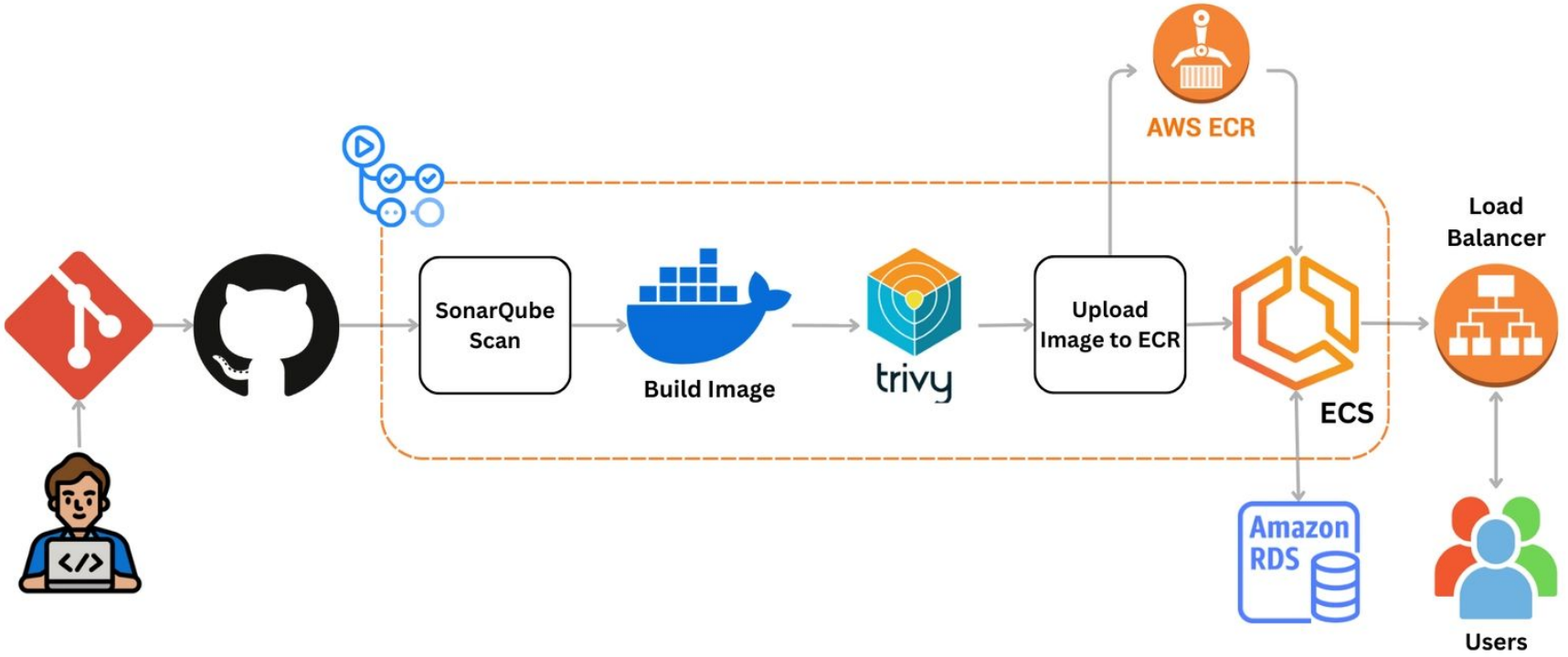
- Reconcile desired state from Git
- Continuously enforce configuration

Audit relevance

- No “manual deploys”
- Git becomes deployment evidence

Common audit red flags

- Pipelines editable without review
- Secrets stored in pipeline YAML
- Manual production deploys via CLI
- No artifact immutability



Dev stores code in local SCM **(Git),** pushes to the central repository **(Github)**

Github actions is configured to run the pipeline after a **commit** is detected. It first runs a static code scan with **Sonar Cloud**, then builds the **docker image**, uses **Trivy** to scan for vulnerabilities and **uploads** the image to **ECR**

ECR picks the **uploaded** image, the pipeline passes the recently **built version** to ECR which then updates its **environment.**

Users access the **application** through a **load balancer**

Analize Pieline

Infrastructure as Code

What Is Infrastructure as Code?

- Infrastructure is defined using code, not manual configuration
- The desired state of infrastructure is:
  - declarative
  - version-controlled
  - reproducible
- Changes are applied through automated workflows

Why IaC Matters

Operational perspective

- Consistent environments (dev = prod)
- Fast provisioning and teardown
- Reduced configuration drift

Security & audit perspective

- Infrastructure changes follow the same controls as application code
- Enables:
    - peer reviews
    - approvals
    - rollback
- Eliminates “click-ops” (manual console changes)

Core IaC Tools

- Terraform — cloud resources
- Helm — application configuration
- Kustomize — environment overlays

Auditability of IaC

What Makes IaC Auditable?

- Version Control
- Plan vs Apply
- Drift Detection

Version Control

Every change is:

- tracked
- attributed to an identity
- timestamped

Audit evidence

- Git commit history
- Pull requests
- Code reviews

Plan vs Apply

- terraform plan shows what will change
- terraform apply executes approved changes

Audit relevance

- Separation between:
  - intent
  - execution
- Strong change-management control

Drift Detection

Detects:

- manual changes
- configuration violations
- Ensures runtime state matches declared state

Audit relevance

- Proof that manual changes are not tolerated
- Evidence of continuous compliance

IaC Security & Compliance Tooling

Tool	Purpose	Typical Use
Checkov	Policy-as-code scanning	Prevent misconfigurations
tfsec	Terraform security scanning	Fast local checks
Terraform Cloud/Enterprise	Policy enforcement	Governance & approvals
OPA / Conftest	Custom compliance rules	Organization-specific policies
Infracost	Cost impact analysis	Financial risk control

Logging, Monitoring and Observability

Observability

Logs, Metrics, Traces

Logs

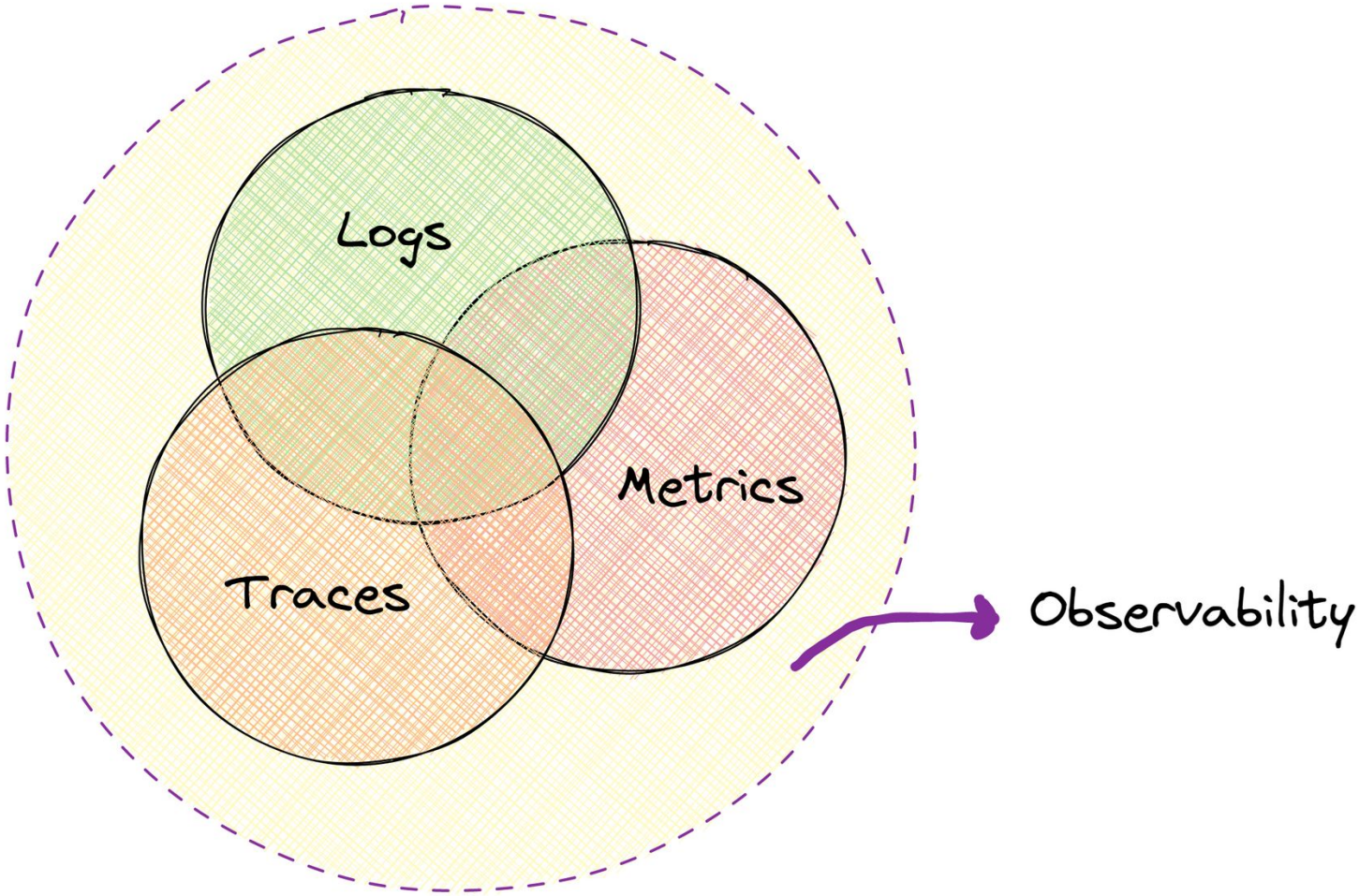
- Discrete, timestamped records of events
- Human-readable or structured
- Best for forensics and investigations

Metrics

- Numeric measurements over time
- Aggregated and efficient
- Best for health, trends, and alerting

Traces

- End-to-end view of a request
- Show how a request flows across services
- Best for root-cause analysis



atatus

Why Observability Matters

Operational view

- Faster incident detection
- Faster recovery
- Reduced mean time to resolution (MTTR)

Security & audit view

- Visibility into:
  - system behavior
  - access patterns
  - failures and anomalies
- Enables evidence based conclusions, not assumptions

Log Retention & Access

Log Retention

What it means

- How long logs are stored
- Where they are stored
- In which form (raw vs aggregated)

Why it matters

- Compliance requirements (e.g. months vs years)
- Ability to investigate incidents after the fact

Log Retention

Audit considerations

- Retention must be:
- defined
- documented
- enforced automatically

Log Access Control

What it means

- Who can:
  - read logs
  - export logs
  - delete logs

Security risks

- Logs often contain:
  - IP addresses
  - user identifiers
  - request metadata
- Logs can be sensitive data

Log Access Control

Audit evidence

- RBAC policies for logging systems
- Access logs for log systems themselves
- Immutable or write-once storage (where required)

Common Audit Red Flags

- Logs stored locally on nodes
- No defined retention period
- Shared admin access to logs
- Logs deletable without trace

Tracing and Metrics: Audit Value

Why Tracing Has Audit Value

Shows how a request moved through the system

- Identifies:
  - which services were involved
  - where failures occurred
  - how long each step took

Audit use cases

- Incident reconstruction
- Data-flow verification
- Proof of segregation between services

Why Metrics Have Audit Value

Provide objective, quantitative evidence

- Answer questions like:
  - Was the system overloaded?
  - Did autoscaling work as expected?
  - Were SLAs violated?

Audit use cases

- Capacity management validation
- DoS and abuse analysis
- Operational risk assessment

Prometheus, Grafana, ELK / EFK Stacks

Prometheus

What it is

- Metrics collection and storage system
- Pull-based model
- Native Kubernetes integration

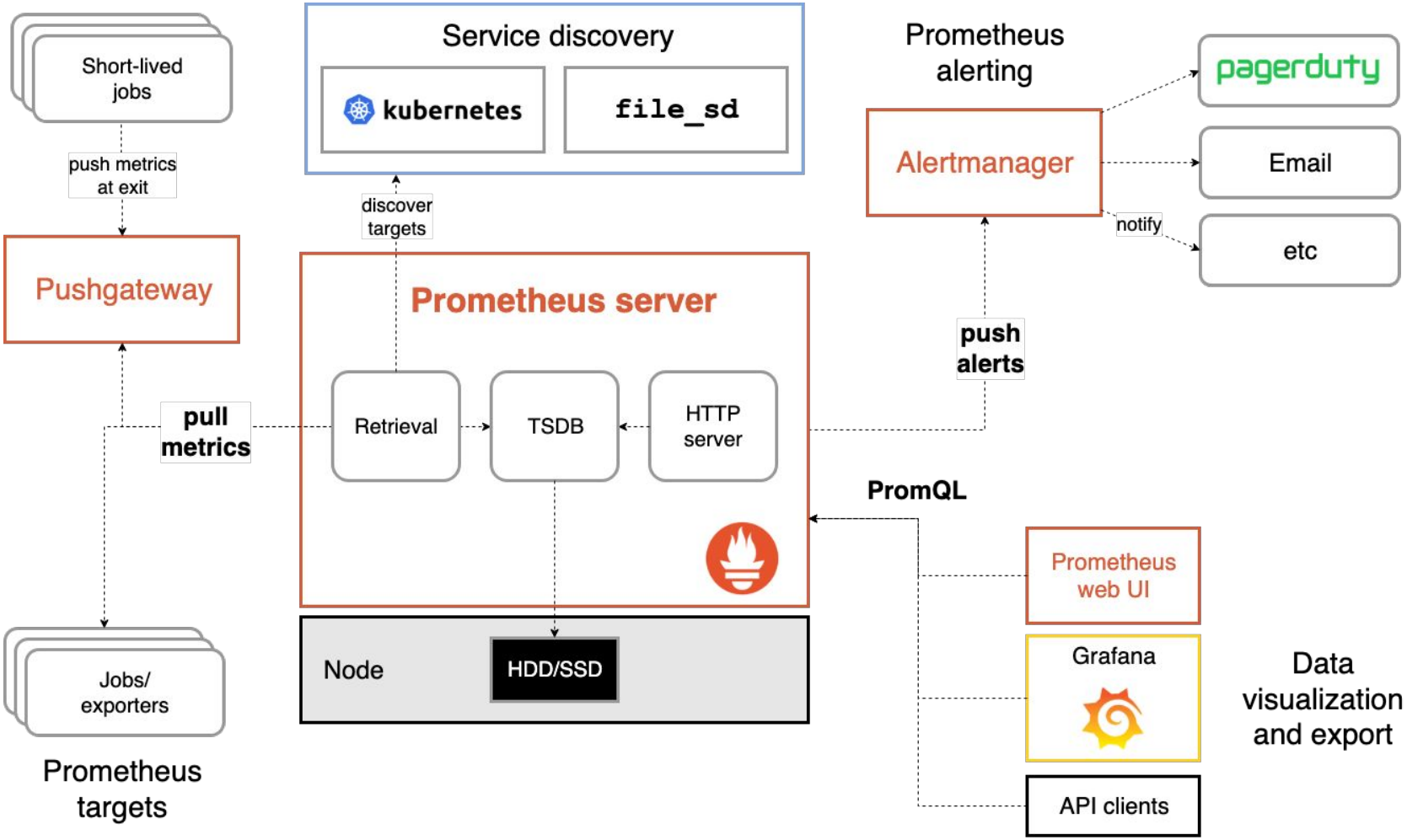
Typical use

- CPU, memory, request rates
- Autoscaling input (HPA)
- Alerting

Prometheus

Audit value

- Historical performance data
- Evidence of capacity controls
- Proof of SLA / SLO monitoring


Short-lived jobs
push metrics at exit
Pushgateway
Service discovery
kubernetes
file_sd
discover targets
Prometheus server
Retrieval
TSDB
HTTP server
pull metrics
Node
HDD/SSD
Jobs/ exporters
Prometheus targets
Prometheus alerting
Alertmanager
pagerduty
Email
etc
notify
push alerts
PromQL
Prometheus web UI
Grafana
API clients
Data visualization and export

Grafana

What it is

- Visualization and dashboarding tool
- Works with Prometheus, Elasticsearch, and more
- Typical use

Dashboards for:

- system health
- application performance
- security signals

Grafana

Audit value

- Read-only dashboards for auditors
- Visual evidence of controls in action
- Historical views for specific incidents


Kubernetes Nginx Ingress Controller NextGen - DevOps Nirvana2
Last 3 hours
1m
Controller Class
All
Namespace
live
Ingress
All
Ingress Pod
All
Config Reloads
Overview
Requests (period)
67.3 K
% Succe...
97.0%
Conns (...
311
Reqs (2...
850
% Success (2m)
99.3%
HTTP 1/2xx (2m)
592
HTTP 3xx (2m)
228
HTTP 4xx (2m)
30
HTTP 5xx (2m)
0
HTTP Requests / Ingress
1.25 K
1 K
750
500
250
0
22:00
22:30
23:00
23:30
00:00
00:30
HTTP Status Codes
1.50 K
1 K
500
0
22:00
22:30
23:00
23:30
00:00
00:30
HTTP 200
HTTP 204
HTTP 206
HTTP 301
HTTP 302
HTTP 304
HTTP 400
HTTP 403
HTTP 404
HTTP 408
HTTP 499
HTTP 500
HTTP 502
HTTP 503
HTTP 504
Total HTTP Requests
3 K
2.50 K
2 K
1.50 K
1 K
500
Latency
Latency (Average Percentiles)
50 s
40 s
30 s
20 s
10 s
0 s
22:00
23:00
00:00
min
max
avg
0.99
4.3 s
42.3 s
6.5 s
0.95
2.4 s
4.5 s
2.8 s
0.9
2.1 s
2.7 s
2.3 s
0.5
357.2 ms
631.6 ms
443.7 ms
0.75
1.0 s
1.8 s
1.5 s
Average
3.2 s
5.1 s
4.1 s
Latency Heatmap
1 min
20 s
5 s
1 s
250 ms
50 ms
10 ms
22:00
22:30
23:00
23:30
00:00
00:30
0
1,000
2,000
2,869
Connections
New Connections Opened (Controller / Ingress Pod)
700
600
500
400
300
200
100
0
Total Connections Open (Controller / Ingress Pod)
315
313
310
308
305
303

ELK / EFK Stack

What it is

- Elasticsearch — log storage and search
- L/F — Logstash or Fluentd — log ingestion
- Kibana — visualization and querying

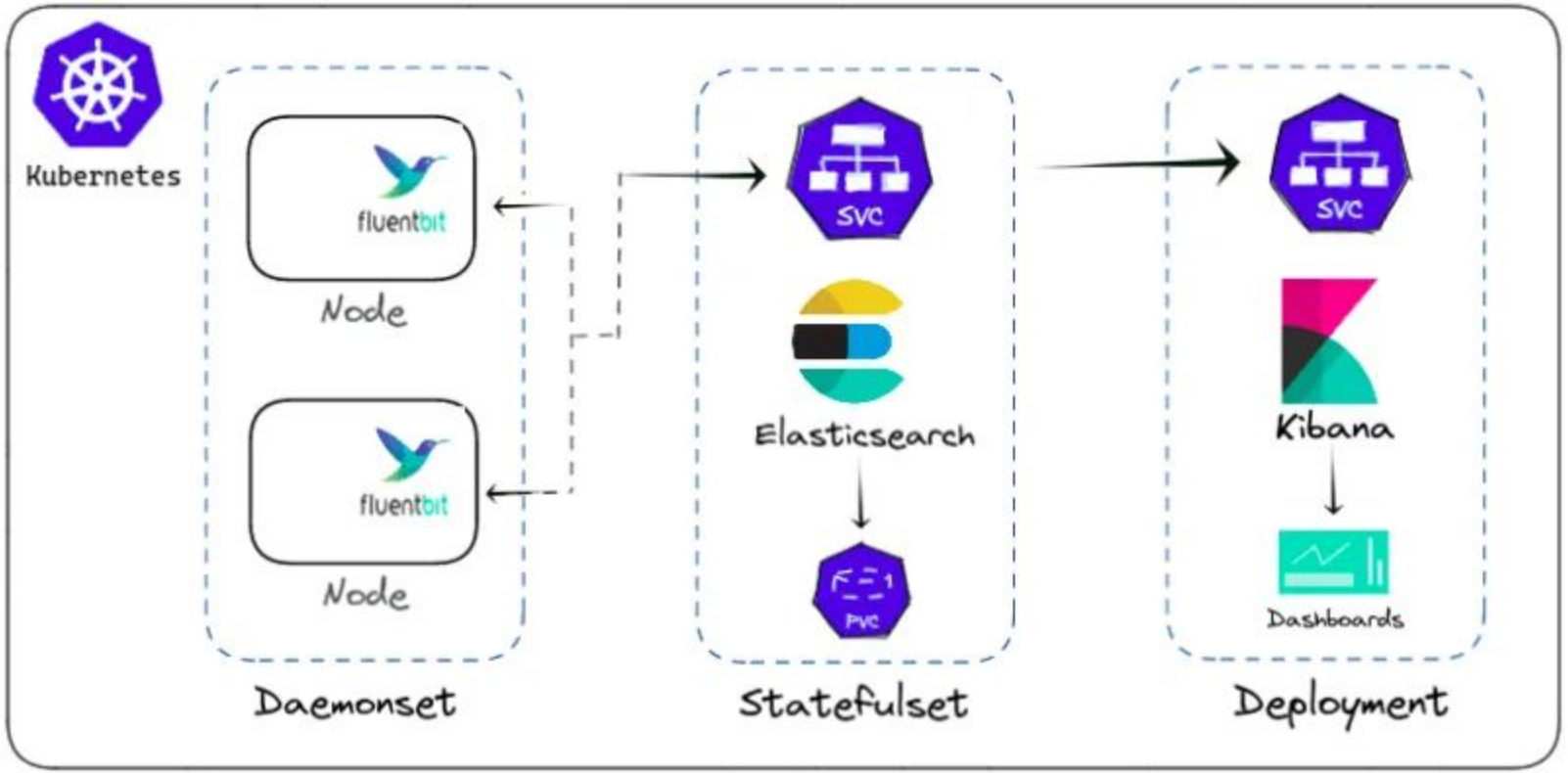
Typical use

- Centralized logging
- Full-text search
- Correlation across systems

ELK / EFK Stack

Audit value

- Forensic investigations
- Long-term log retention
- Advanced filtering and correlation


Kubernetes
fluentbit
Node
fluentbit
Node
Daemonset
SVC
Elasticsearch
PVC
Statefulset
SVC
Kibana
Dashboards
Deployment

Security in Cloud Native Environments

Kubernetes RBAC

What is Kubernetes RBAC?

- RBAC (Role-Based Access Control) defines who can do what in a Kubernetes cluster
- Permissions are granted via:
  - Roles / ClusterRoles (what actions are allowed)
  - RoleBindings / ClusterRoleBindings (who gets those permissions)

Core RBAC Objects

- Role — permissions within a namespace
- ClusterRole — cluster-wide permissions
- RoleBinding — binds a Role to a user/service account
- ClusterRoleBinding — binds a ClusterRole globally

Why RBAC Matters

Security perspective

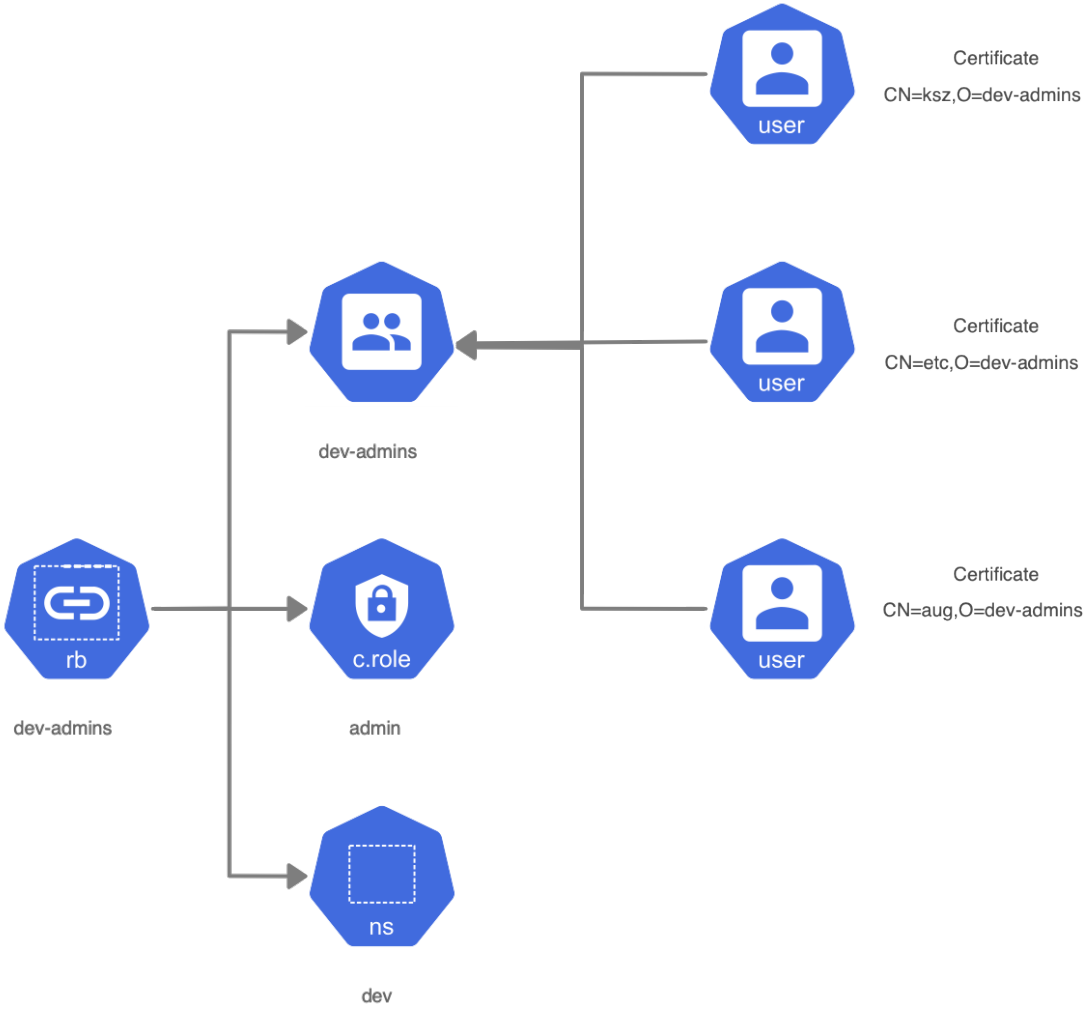
- Enforces least privilege
- Limits blast radius of compromised identities
- Separates human and workload access

Audit perspective

- Clear, declarative access rules
- Easy to answer:
  - Who can deploy?
  - Who can read secrets?
- RBAC configs = direct audit evidence

Common RBAC Anti-Patterns

- cluster-admin used by applications
- Shared service accounts
- RBAC changes outside Git
- No periodic access reviews


user
Certificate
CN=ksz,O=dev-admins
user
Certificate
CN=etc,O=dev-admins
user
Certificate
CN=aug,O=dev-admins
dev-admins
rb
dev-admins
c.role
admin
ns
dev

Secret Management

Kubernetes Secrets (Native)

- Store sensitive data (tokens, passwords, keys)
- Mounted as:
  - environment variables
  - files
- Base64-encoded, not encrypted by default

External Secret Management

- Centralized secret storage
- Dynamic secrets (short-lived credentials)
- Automatic rotation and revocation
- Strong audit logging

Audit Questions to Ask

- Where are secrets stored?
- Who can access them?
- How often are they rotated?
- Can access be revoked immediately?

External Secret Management Integration

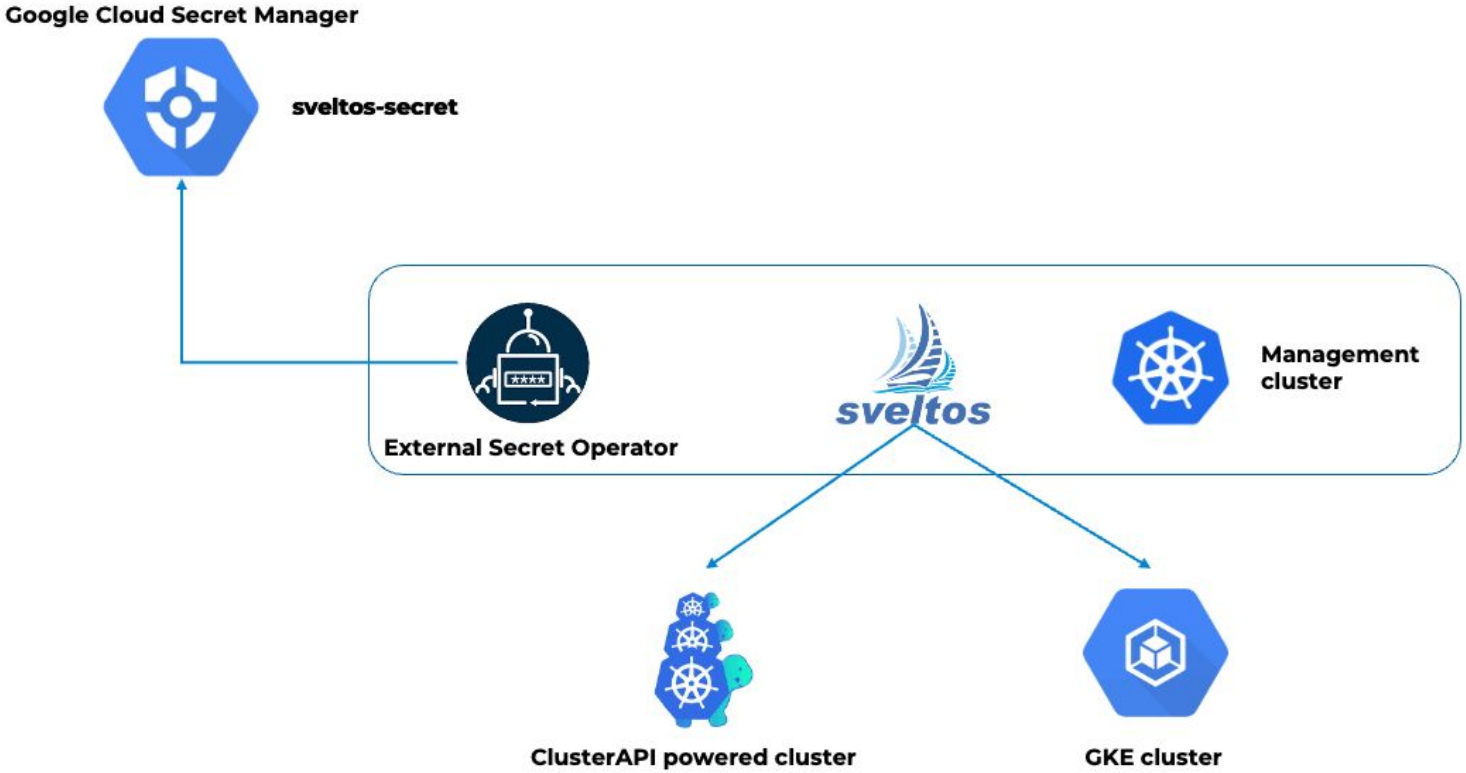


Image scanning, digital signatures, SBOMs

Image Scanning

Scans container images for:

- known vulnerabilities (CVEs)
- outdated libraries
- Typically integrated into CI pipelines

Audit value

- Proof that images are checked before deployment
- Risk-based acceptance of vulnerabilities

Digital Signatures (Image Signing)

- Cryptographically sign container images
- Verify image integrity and origin at deploy time

Why it matters

- Prevents tampering
- Ensures only trusted images run in production

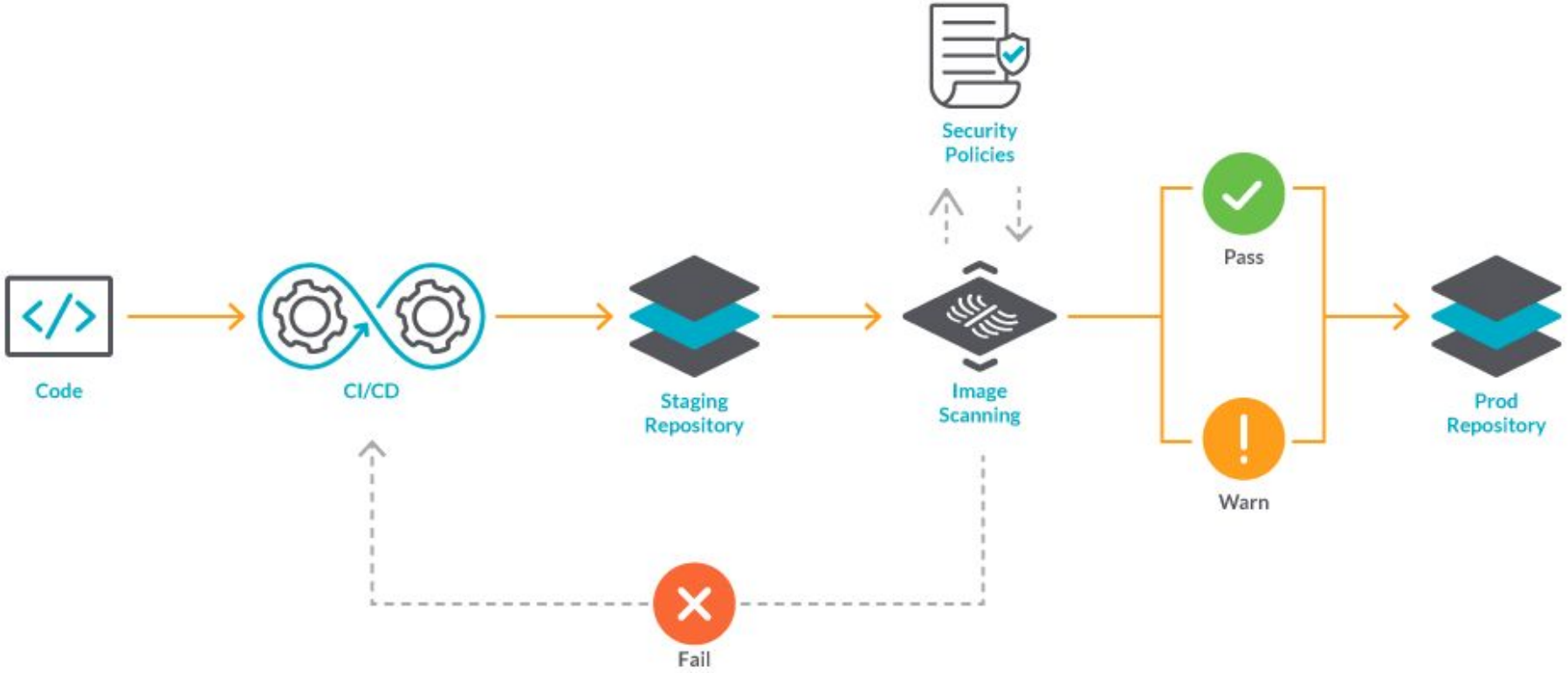
SBOM (Software Bill of Materials)

Machine-readable list of:

- libraries
- dependencies
- versions
- Generated at build time

Audit value

- Transparency of software composition
- Faster incident response (e.g. Log4Shell)
- Regulatory compliance support


Security Policies
Code
CI/CD
Staging Repository
Image Scanning
Pass
Warn
Prod Repository
Fail

Supply chain security

What Is Supply Chain Security?

- Protects the entire path:
  - source code
  - build process
  - artifacts
  - deployment
- Assumes attackers may target CI/CD systems, not just apps

Software Supply Chain Security

Developer

Source

Managing access to IDEs

Source code review

Source code management (SCM)

Build

CI/CD Pipeline

Containers & Registries

Dependencies

Package & Deploy

SBOMs

Code provenance & signature

Artifact repository

SLSA (Supply-chain Levels for Software Artifacts)

- A framework defining maturity levels for build security
- Focuses on:
  - build provenance
  - reproducibility
  - tamper resistance

Audit relevance

- Provides a measurable security maturity model
- Answers: How trustworthy is this artifact?

in-toto

Framework for securing the software supply chain

- Verifies:
  - each step in the build pipeline
  - who performed it
  - what was produced

Audit value

- End-to-end traceability
- Strong non-repudiation of build steps

Why Auditors Care

- Many modern attacks bypass runtime security
- Supply chain controls answer:
  - Who built this?
  - How was it built?
  - Has it been modified?

Training Summary — What We Learned

- During this training, we learned:
- how cloud computing shifts security and control from manual operations to identity, policies, and APIs
- how cloud-native and microservice architectures, supported by 12-Factor App principles, improve isolation, resilience, and auditability
- how containers and Kubernetes manage applications through declarative, self-healing mechanisms
- how Deployments, probes, autoscaling (HPA/VPA), and rollouts enable controlled, auditable application lifecycle management
- how CI/CD pipelines and Infrastructure as Code turn change management into repeatable, reviewable, and enforceable processes
- how observability (logs, metrics, traces) provides evidence for incident analysis and compliance
- how Kubernetes RBAC and secret management enforce least privilege and protect sensitive data
- how image scanning, SBOMs, digital signatures, and supply-chain security frameworks (SLSA, in-toto) help ensure that only trusted software reaches production

Final takeaway

Cloud-native security is not about adding more tools — it is about designing systems that generate control and audit evidence by default.