le cnam

Software supply chain security for the cloud

Cloud virtualization stack - 2

Jacopo Bufalino (CNAM)

Recap from previous lecture

- ➜ VMs are slow
- ➜ Hard to maintain
- ➜ Do not scale very well
- ➜ Containers are fast, easy to share and Immutable
  - ▶ Solve the problem of “But it works on my machine”

Microservice applications

Containers for the service oriented applications

Containers provide lightweight, portable environments for deploying and managing service-oriented applications (SOA). They allow to effectivey split appliations into small services that can be deployed independently.

Microservices Architecture

Microservices architecture is an approach to building applications as a collection of small, independent services.

Microservices principles

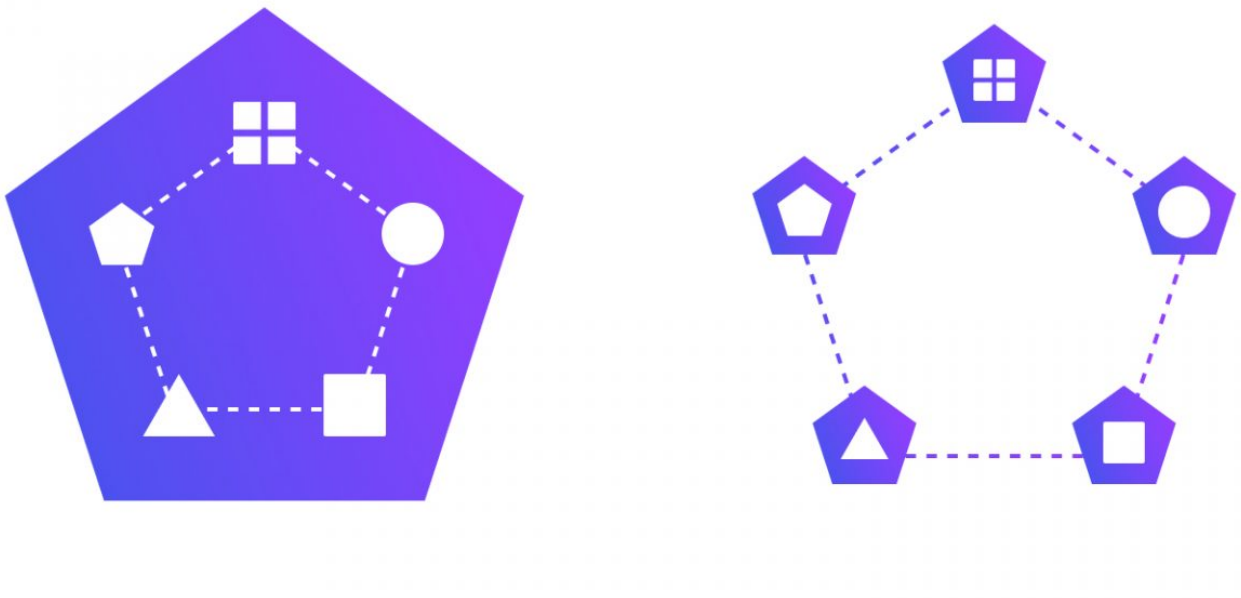
Each service is:

- **Independently deployable**: Can be updated without affecting others
- **Loosely coupled**: Minimal dependencies between services
- **Organized around business capabilities**: Represents a specific function

Contrast with monolithic architecture

Instead of one large application, break it into many small services that communicate over network protocols.

Microservices



Microservices, Deduplication and Common Issues

- Microservices split applications into smaller independent services
- Each service owns its business logic data and deployment lifecycle

Advantages

- Independent scaling: saves in costs
- Faster deployments: less downtime
- Technology flexibility: right tool for the job

Issues

- How to handle cross-cutting concerns like security and monitoring?
- Handling multi-site and complex scaling

Limitations of container runtimes

Recall last lecture: *container runtimes automate the lifecycle of individual containers*.

What container runtimes DO handle

- ➜ Starting and stopping containers
- ➜ Managing container processes
- ➜ Enforcing resource limits (CPU, memory)
- ➜ Isolating containers using namespaces and cgroups

What they DON'T handle

Everything needed to run group of containers at scale and across hosts

In practice

Managing 5 containers:

- ✓ Restart failing containers
- ✓ Edit configuration by hand
- ✓ Re-run failing commands

Managing 500 containers:

- ✗ Hours of manual work
- ✗ Error-prone
- ✗ Impossible to maintain

Example

A typical cloud application has 20-50 containers, each with 3-10 replicas across multiple hosts.

Limitation 1: Storage Management

Container runtimes support volumes, but lack distributed storage orchestration.

Manual operations required

- ➜ Managing volume lifecycle (creation, deletion, backup)
- ➜ Ensuring data availability when containers move between hosts
- ➜ Handling volume snapshots and disaster recovery
- ➜ Coordinating shared storage across multiple containers

Limitation 2: Networking Complexity

Container runtimes provide only basic networking.

Manual operations required

- ➜ Assigning IP addresses to each container
- ➜ Configuring DNS resolution between services
- ➜ Setting up load balancers for traffic distribution
- ➜ Managing network routing across multiple hosts
- ➜ Updating all dependent services when IPs change

Networking example

One frontend and three backends (A,B, C)

Before restart:

- Backend-A: 172.17.0.2
- Backend-B: 172.17.0.3
- Backend-C: 172.17.0.4

Frontend configured to use all three IPs

After Backend-B crashes:

- Backend-A: 172.17.0.2
- Backend-B: 172.17.0.9 (new!)
- Backend-C: 172.17.0.4

Frontend still tries 172.17.0.3 $\rightarrow$ fails

Limitation 3: Scaling Challenges

Container runtimes can start containers, but cannot intelligently scale them.

Manual operations required

- ➜ Monitoring application metrics (CPU, memory, request rate)
- ➜ Deciding when to scale up or down
- ➜ Choosing which host has available capacity
- ➜ Starting new container replicas manually
- ➜ Registering new instances with load balancers
- ➜ Removing excess containers during low traffic

Limitation 4: Provisioning and Placement

Manual operations required

- ➜ Determining which host has sufficient resources
- ➜ Balancing workloads across multiple machines
- ➜ Handling host failures and relocating containers

Resource waste

Without intelligent placement, some hosts will be under-utilized while others are overloaded. This wastes money and causes performance problems.

Summary: what do we still need to run containerized apps at scale

Container runtimes manage individual containers on a single host.
But production applications require:

- ➜ **Distributed storage** that follows containers across hosts
- ➜ **Service discovery** and stable networking abstractions
- ➜ **Automated scaling** based on real-time metrics
- ➜ **Intelligent scheduling** across clusters of machines
- ➜ **Self-healing** without human intervention

Container Orchestrators

Container Orchestrators

Orchestrators are distributed systems that coordiate multiple containers on the same host and across different hosts.

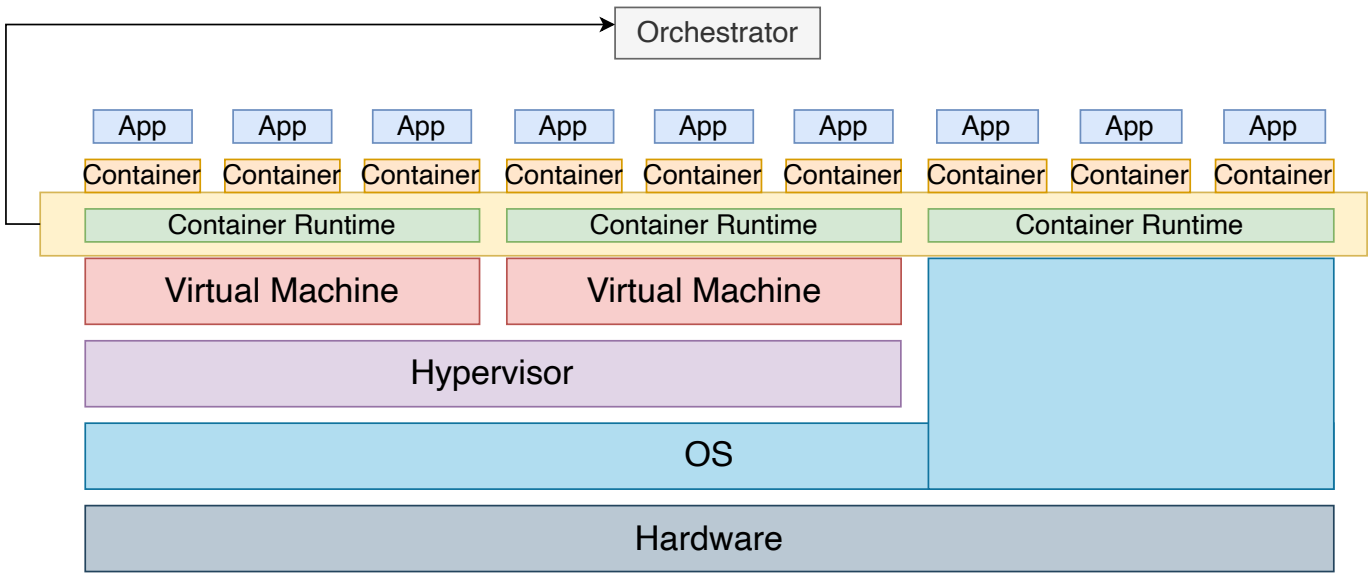
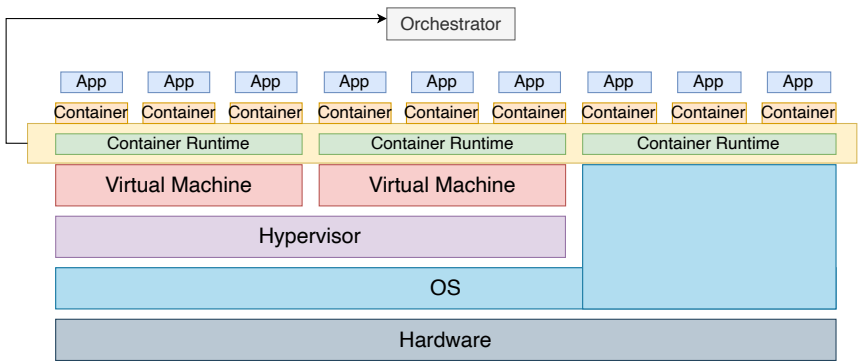


Figure: Container Orchestrator Overview

Container Orchestrators: Key Features

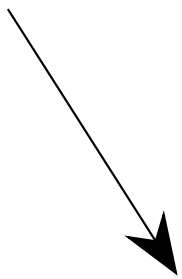
- ➜ Declarative configuration
- ➜ Scheduling and scalability across clusters of nodes
- ➜ Advanced service discovery (DNS) and load balancing



Declarative configurations

Listing 1: Imperative language

```
docker run -d nginx:1.14.2
docker run -d nginx:1.14.2
docker run -d nginx:1.14.2
docker run -d -n nginx-lb -p 80:80
```



Listing 2: Declarative Language

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
      - name: nginx
        image: nginx:1.14.2
```

Declarative configurations

- ➜ Simplifies management by defining a desired state
- ➜ Enhances version control and auditing of configurations
  - Security?
- ➜ Consistency and reproducibility
- ➜ Idempotent deployments
  - Applying the same configuration multiple times results in the same final state.

Deployment Models of Container Orchestrators

Container orchestrators can be deployed in different models:

- **On-Premises:** Installed in enterprise data centers for full control.
- **Cloud-Based:** Offered as managed services by providers (e.g., GKE, EKS, AKS).
- **Hybrid:** Combines on-premises infrastructure with cloud services.

On-Premises Deployment

Self-managed Kubernetes clusters running in enterprise data centers or private infrastructure.

Advantages:

- ➜ Complete infrastructure control
- ➜ Data sovereignty guaranteed
- ➜ No cloud vendor lock-in
- ➜ Predictable costs
- ➜ Network latency control

Disadvantages:

- ➜ High investment
- ➜ Requires specialized expertise
- ➜ Manual upgrades and patching
- ➜ Limited scalability
- ➜ Higher maintenance burden

Cloud-Based (Managed)

Cloud providers offer container orchestration

Major managed services

- **Google Kubernetes Engine (GKE)**
- **Amazon Elastic Kubernetes Service (EKS)**
- **Amazon Elastic Container Service (ECS)**
- **OpenShift**

Cloud-Based: Advantages

Operational benefits:

- ➜ No control plane management
- ➜ Automatic upgrades available
- ➜ Built-in HA and redundancy
- ➜ Elastic scalability
- ➜ Faster time-to-production

Cost benefits:

- ➜ Pay-as-you-go (PaaS)
- ➜ Reduced staff requirements
- ➜ Auto-scaling reduces waste
- ➜ Less debugging
- ➜ Free control plane (some providers)

Cloud-Based: Potential drawbacks

- **Vendor lock-in**: Cloud-specific integrations and features
- **Cost unpredictability**: Can escalate with traffic/storage growth
- **Data residency**: Data may cross geographic boundaries
- **Limited control**: Cannot customize control plane configuration
- **Network costs**: Data transfer fees can be substantial
- **Compliance challenges**: Some regulations prohibit cloud hosting

Hybrid Deployment Model

Hybrid architectures combine on-premises infrastructure with cloud resources.

Common hybrid patterns

- **Cloud bursting**: On-prem primary, cloud for overflow capacity
- **Data locality**: Compute in cloud, sensitive data on-premises
- **DR/Backup**: Production on-prem, disaster recovery in cloud
- **Migration path**: Gradual transition from on-prem to cloud
- **Edge + Core**: Edge processing on-prem, central processing in cloud

Multi-Cloud Deployment

Multi-cloud strategies use multiple cloud providers (no on-premises component).

Motivations for multi-cloud

- **Avoid vendor lock-in**: Not dependent on single provider
- **Geographic coverage**: Use best provider per region
- **Cost optimization**: Leverage competitive pricing
- **Resilience**: Reduce impact of provider outages
- **Compliance**: Meet data residency requirements per country

Common Orchestrators Patterns

Orchestrators enable new architectural patterns by simplifying container connectivity and management.

Key observations

- ➜ Orchestrators allow easy deployment of containerized applications.
- ➜ Adding functionality via new containers is often simpler than code changes
- ➜ New patterns emerge for cross-cutting concerns

Sidecar Pattern

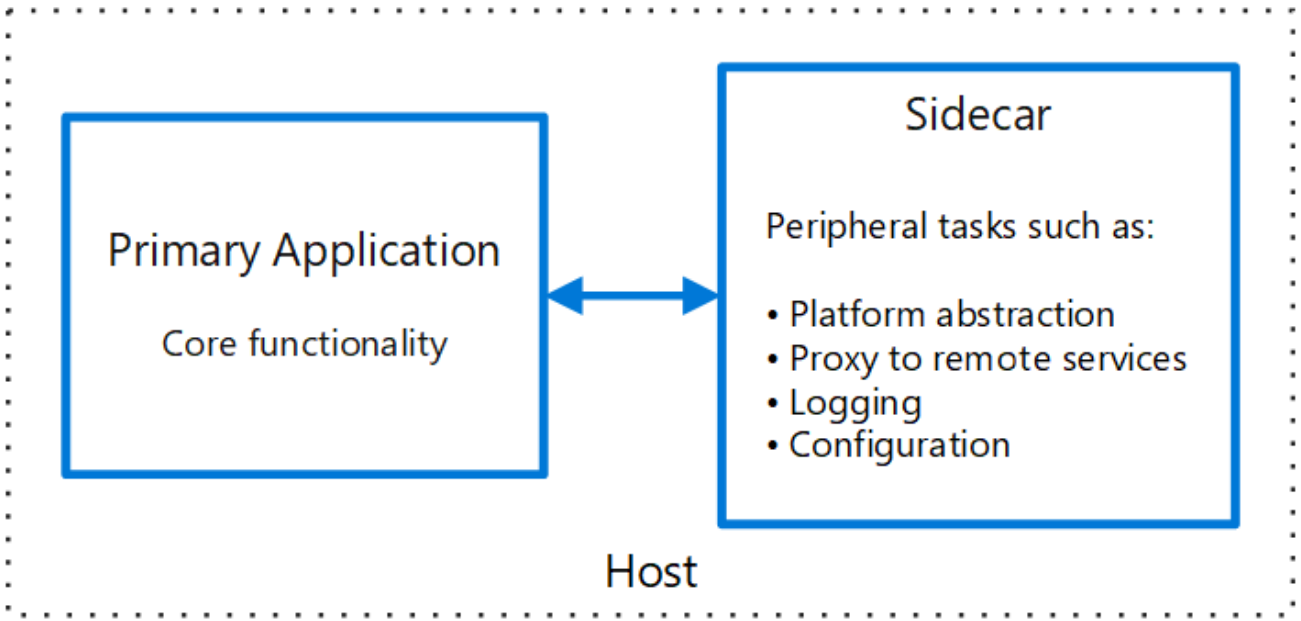
Applications need non-function requirements:

- ➜ Logging
- ➜ Monitoring
- ➜ Authentication
- ➜ Authorization

Complex and time consuming to add them to each application, especially when using different programming languages.

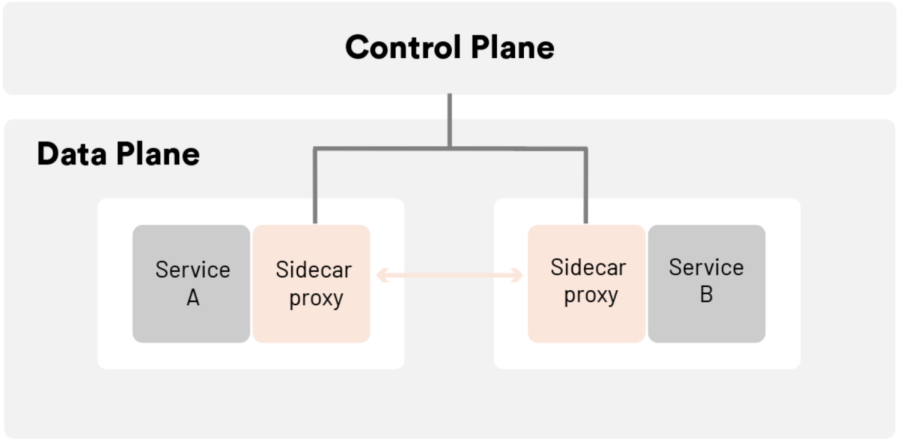
Sidecar pattern - part 2

How to add features without writing new code?



[0] https://learn.microsoft.com/en-us/azure/architecture/patterns/sidecar

Service Mesh



- One sidecar per application
- Sidecar acts as a proxy from/to the applications
- Separate control-plane and data-plane
  - Authorization logic
  - Traffic management
- Useful to centrally manage applications

[0] https://tetrate.io/

Zero Trust

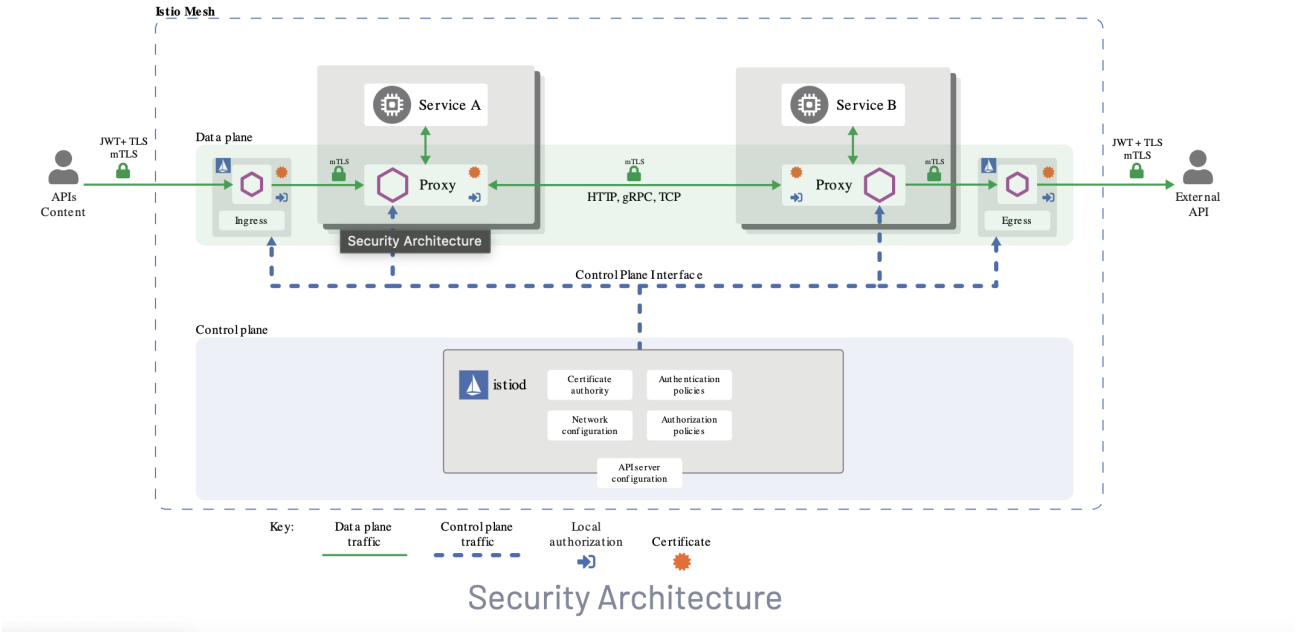


Figure: Source: istio.io

- **Never Trust, Always Verify:** Continuously authenticate and authorize every user, device, and network flow.
- **Least Privilege Access:** Grant minimal access rights necessary for users and systems to perform their tasks

Docker compose

Docker compose: an intermediate solution

- A tool for defining and running multi-container Docker applications
- Uses a YAML file to configure connectivity and runtime (very good!)
- Management of multiple containers on a single host
- Supports scaling, networking, and volume management

Example of a docker compose file

```
services:
  web:
    image: nginx:latest
    container_name: web-server # Hostname
    networks:
      custom-network:
        ipv4_address: 172.20.0.2 # IP
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf # Volume mapping
    ports:
      - "80:80" # Port mapping
    deploy:
      mode: "replicated"
      replicas: 5 # Replication
    restart: unless-stopped # Automatic restart


  database:
    image: mysql:8.0
    container_name: mysql-db
    networks:
      custom-network:
        ipv4_address: 172.20.0.3
    environment:
      MYSQL_ROOT_PASSWORD: password

networks:
  custom-network:
    driver: bridge
    ipam:
      config:
        - subnet: 172.20.0.0/16
          gateway: 172.20.0.1
```

Limitations of Docker Compose

- ➜ Designed for single-host environments
- ➜ Not suitable for production-scale deployments
- ➜ Lacks advanced orchestration features (e.g., service discovery, auto-scaling)
- ➜ No built-in support for cluster management or cloud-native features

Kubernetes

Kubernetes

Kubernetes is the most popular orchestrator for containerized cloud applications.

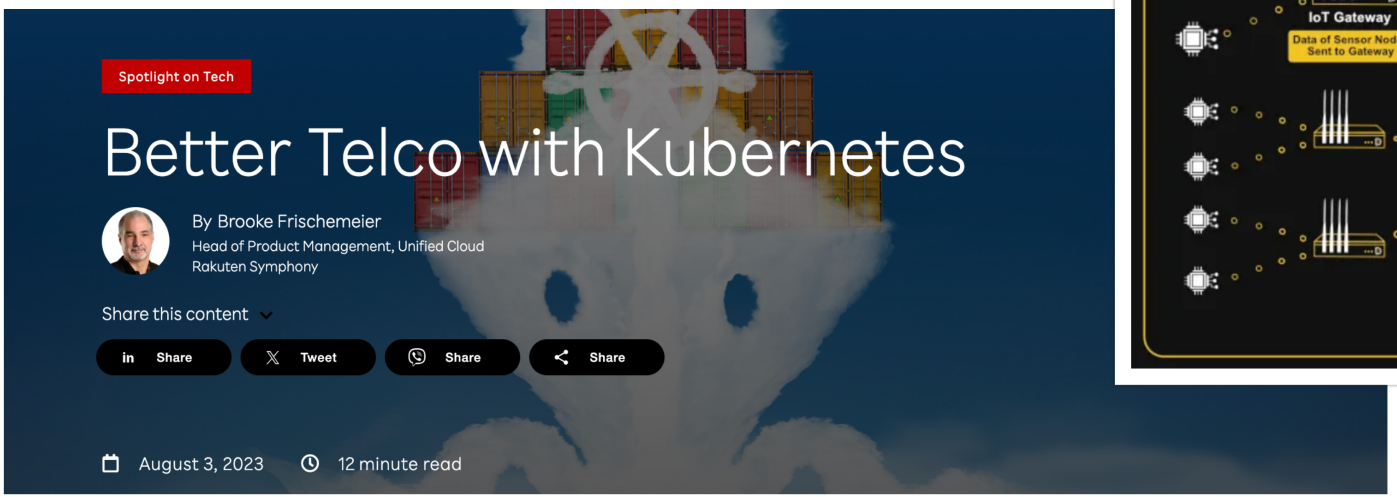
Its popularity is mainly due to

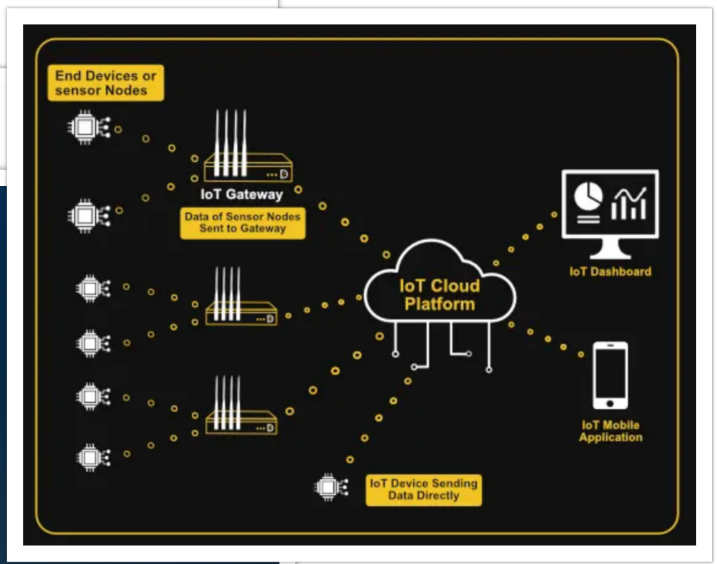
- ➜ Customizability: pluggable modules and interfaces
- ➜ Open source code
- ➜ Rich set of features (e.g., TLS certificate management, secrets, load-balancing)

Use cases

Enhancing Patient Care Through Kubernetes-Powered Healthcare Data Management

U-2 Federal Lab achieves flight with Kubernetes





Architecture

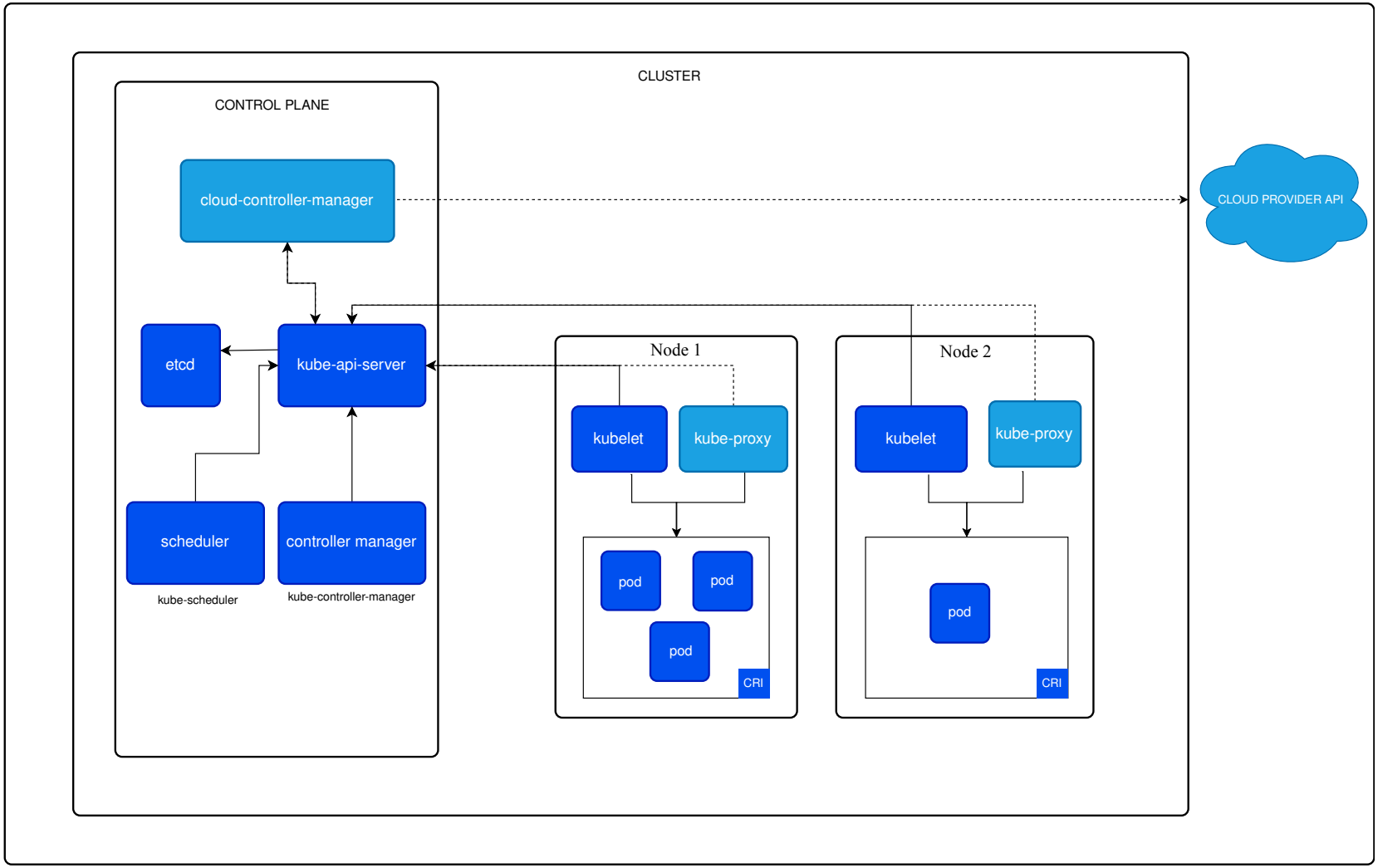


Figure: Kubernetes Cluster Architecture

Interacting with the Kubernetes Cluster

Users and applications interact with Kubernetes through the API Server, which is the central control plane component.

Interaction methods

- **kubectl**: Command-line tool (most common)
- **Dashboards**: Web-based/Desktop/Mobile UI
- **REST API**: Direct HTTPs requests to API Server
- **Client Libraries**: Go, Python, Java, JavaScript, etc.

API Server

Every interaction method ultimately communicates with the Kubernetes API Server via HTTPs.

Authentication and Authorization

Access to Kubernetes API is controlled by authentication and authorization.

Authentication methods

- **X.509 Client Certificates**: Most common for users
- **Bearer Tokens**: ServiceAccount tokens, OIDC tokens
- **Basic Auth**: Username/password (deprecated)
- **OIDC**: Integration with identity providers (Google, Azure AD)

Authorization (what you can do)

- **RBAC**: Role-Based Access Control (most common)
- **ABAC**: Attribute-Based Access Control
- **Webhook**: External authorization service

One tool, different flavours

Kubernetes is a single orchestration platform, but is open to different implementation. It heavily relies on interfaces:

- ➜ Interface CRI: Container Runtime interface
- ➜ Interface CNI: Container Network Interface
- ➜ Interface CSI: Container Storage Interface

CRI: Container Runtime Interface

CRI is a standardizes communication between the kubelet and container runtimes.

Two main tasks

1. Manage container lifecycle
   - Create/start/stop containers
   - Execute commands (exec, attach)
   - Port forwarding
2. Handle container images
   - Pull and list images
   - Image caching and storage
   - Remove unused images

CNI: Container Network Interface

CNI standardizes how network plugins configure networking for containers.

CNI responsibilities

- ➜ Assign IP addresses to pods
- ➜ Configure network interfaces
- ➜ Set up routing rules
- ➜ Enable pod-to-pod communication
- ➜ Enforce network policies

Popular CNI plugins

- ➜ Calico: Network policies + routing
- ➜ Cilium: eBPF-based, high performance
- ➜ Flannel: Simple overlay network (Too simple)

CSI: Container Storage Interface

CSI enables storage vendors to develop plugins without modifying Kubernetes core.

CSI capabilities

- ➜ Provision/delete persistent volumes
- ➜ Attach/detach volumes to nodes
- ➜ Mount/unmount volumes to pods
- ➜ Snapshot and clone volumes
- ➜ Volume expansion

CSI drivers

- ➜ Cloud providers: AWS EBS, Azure Disk, GCP PD
- ➜ Network storage: NFS, Ceph, GlusterFS, SMB
- ➜ Local storage: OpenEBS, Longhorn

The Power of Interfaces

Kubernetes extensibility through interfaces

- **CRI**: Choose runtime based on security/performance needs
- **CNI**: Select networking solution for your architecture
- **CSI**: Use any storage backend without vendor lock-in

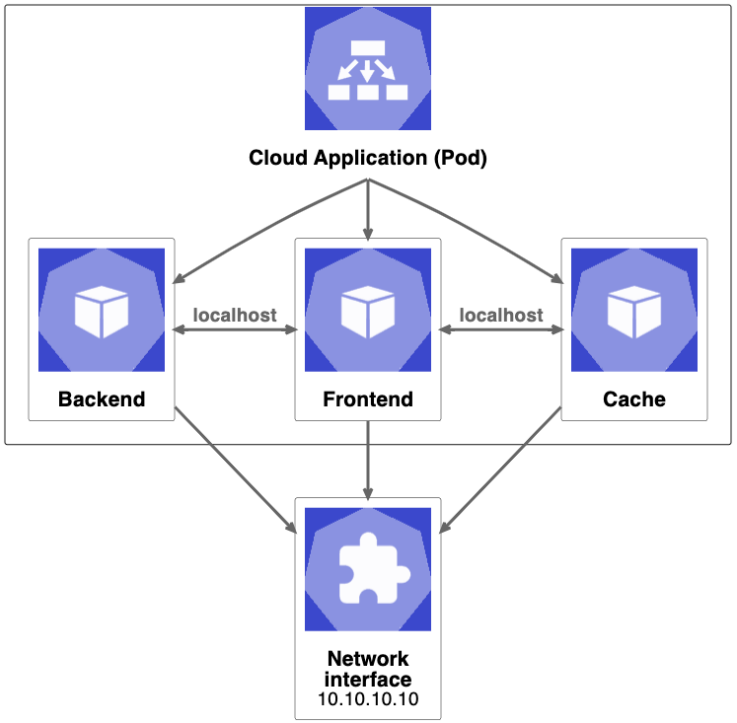
Workloads

A workload is an application running on Kubernetes. The minimal deployable unit is the Pod.

Pods

Pods are groups of containers

- ➜ Tightly-coupled
- ➜ Common lifecycle
- ➜ Same Linux Namespaces



Pods are short-lived ephemeral entities

They can be created and destroyed at any time.

Workloads

Other types of workloads add functionalities to Pods.

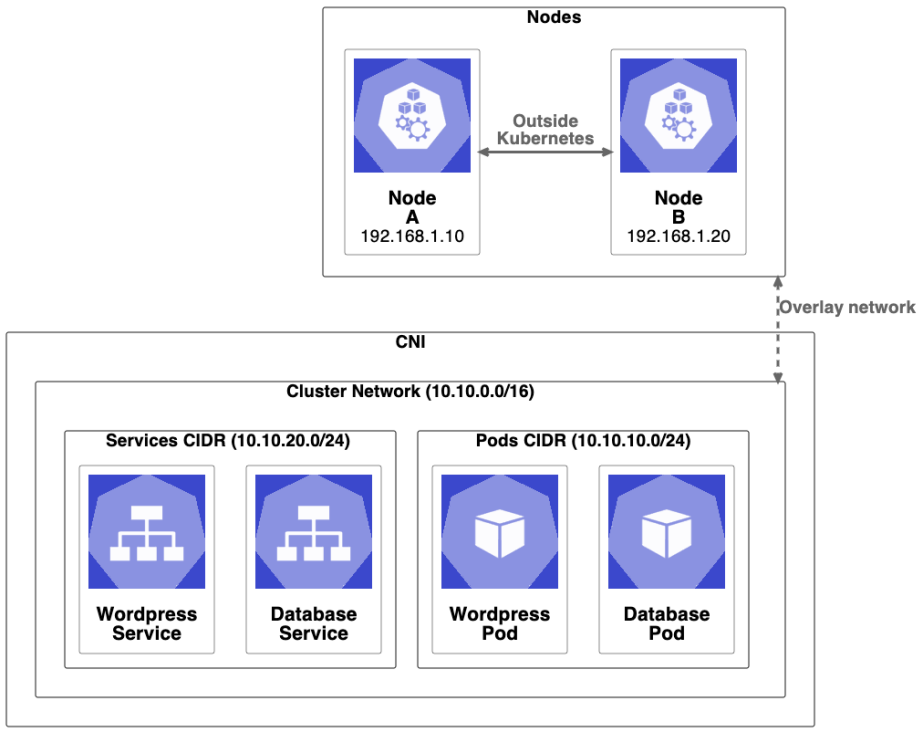
Examples

- **Deployment:** replicate Pods and manage different versions of applications. Also ensure that Pods are re-created if they fail.
- **CronJobs:** run a Pod at given time intervals.

Networking

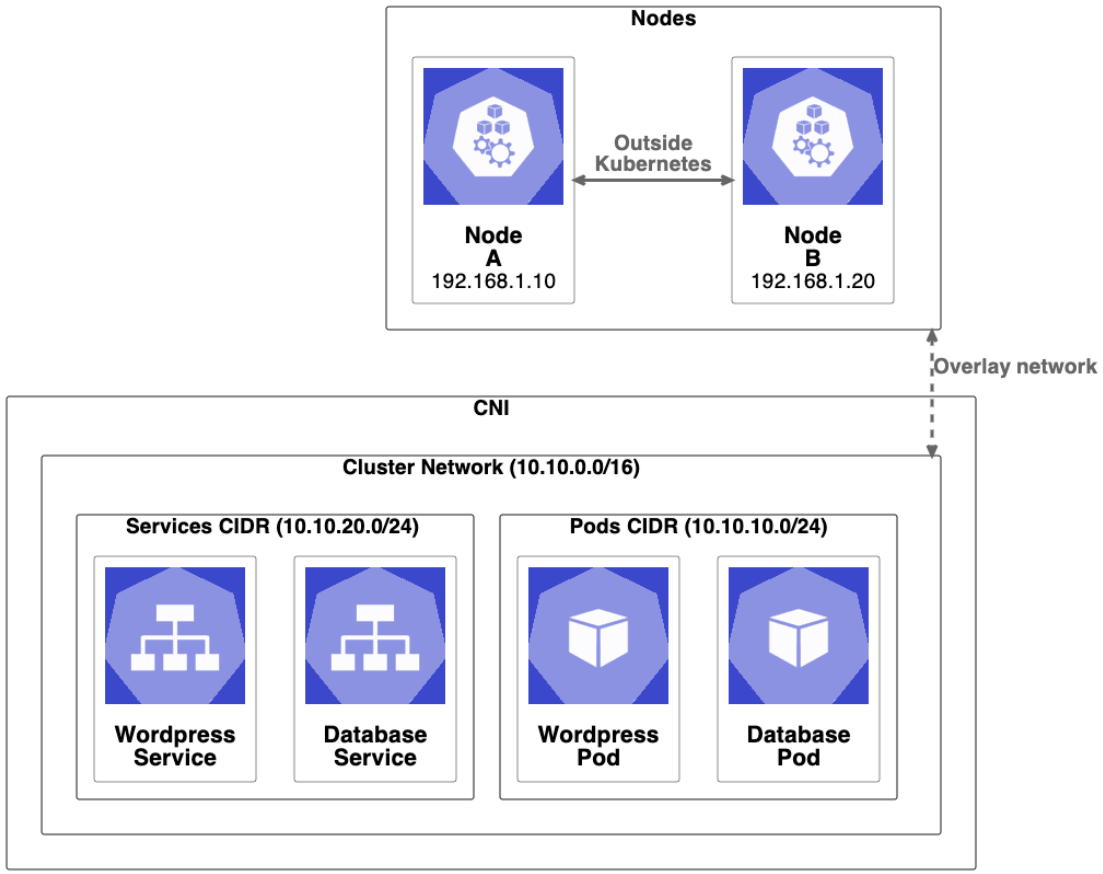
There are different networks:

- Nodes Network
- Containers Network
- Cluster Network
  - Service Network
  - Pods Network



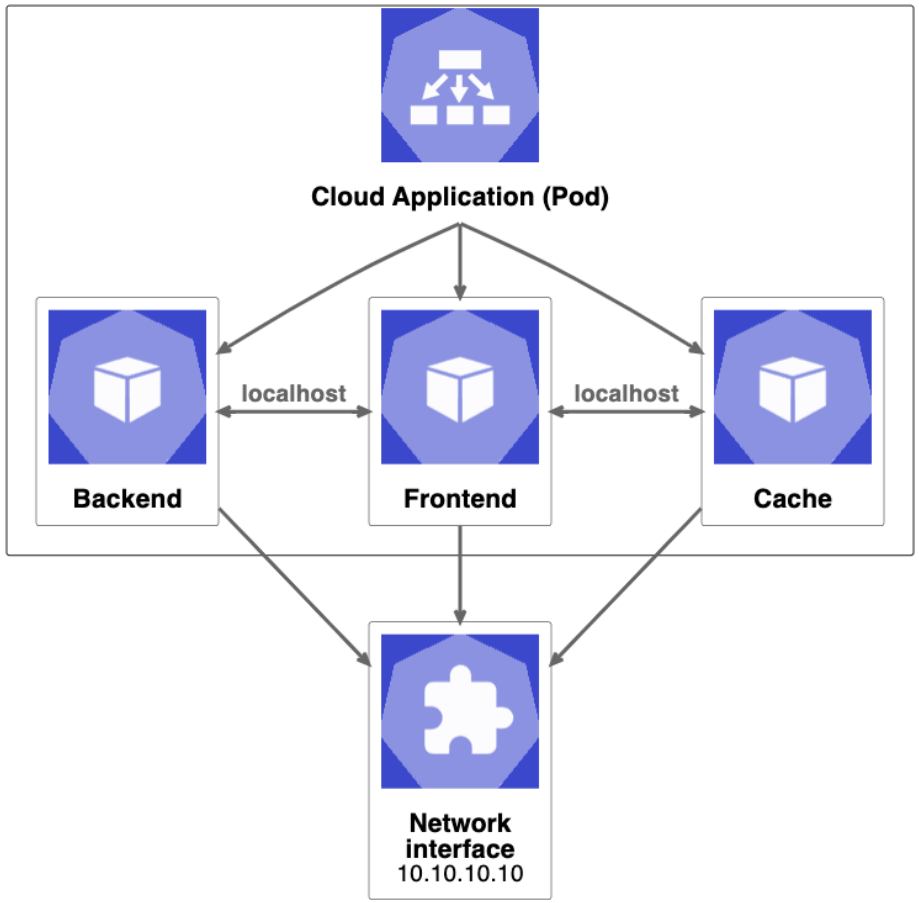
Nodes Network

Nodes in Kubernetes need to be reachable. Kubernetes is responsible for securing their communication by e.g., using mTLS.



Containers network

- A Pod has several containers
- The same Linux Namespaces
  - Single network namespace (and interface)
  - Same IP address



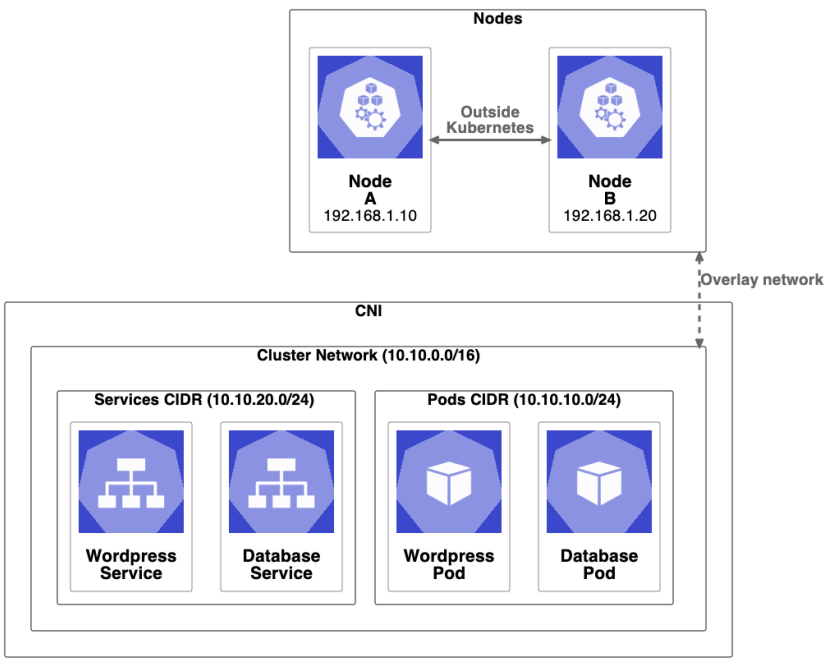
Pods are short-lived entities

They can be created and destroyed at any time. IP addresses are not preserved across restarts.

Cluster Network

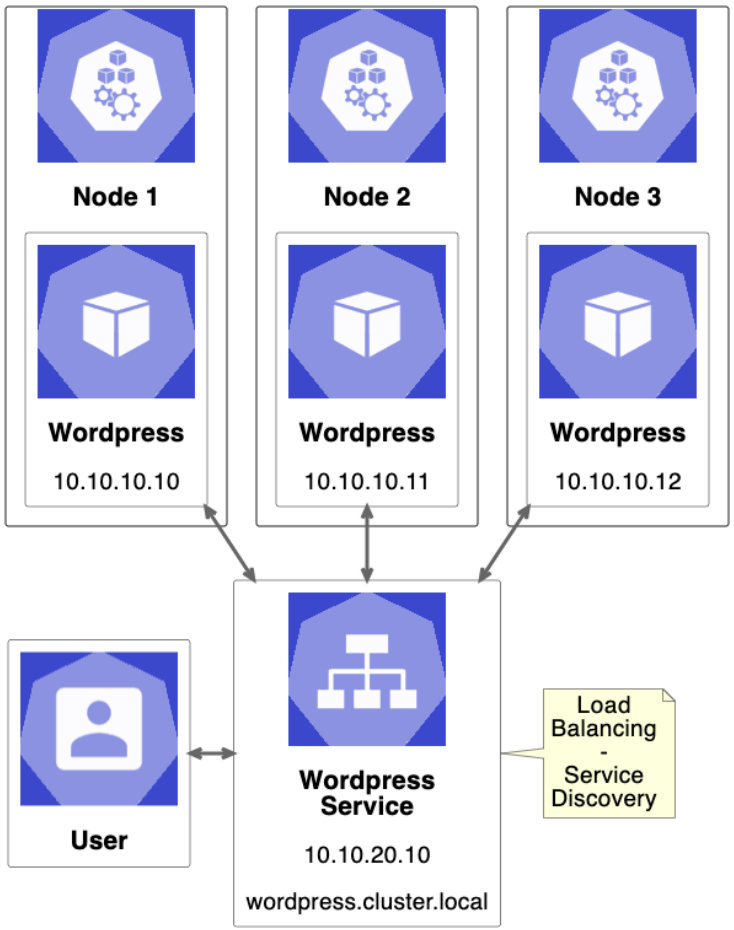
The cluster network is managed by the CNI (Container Network Interface) plugin. Being an interface, it allows for different implementations. The CNI is responsible for:

- ➜ Assigning IP addresses to Pods and Services
- ➜ Managing the overlay network between nodes
- ➜ Enforce network security policies



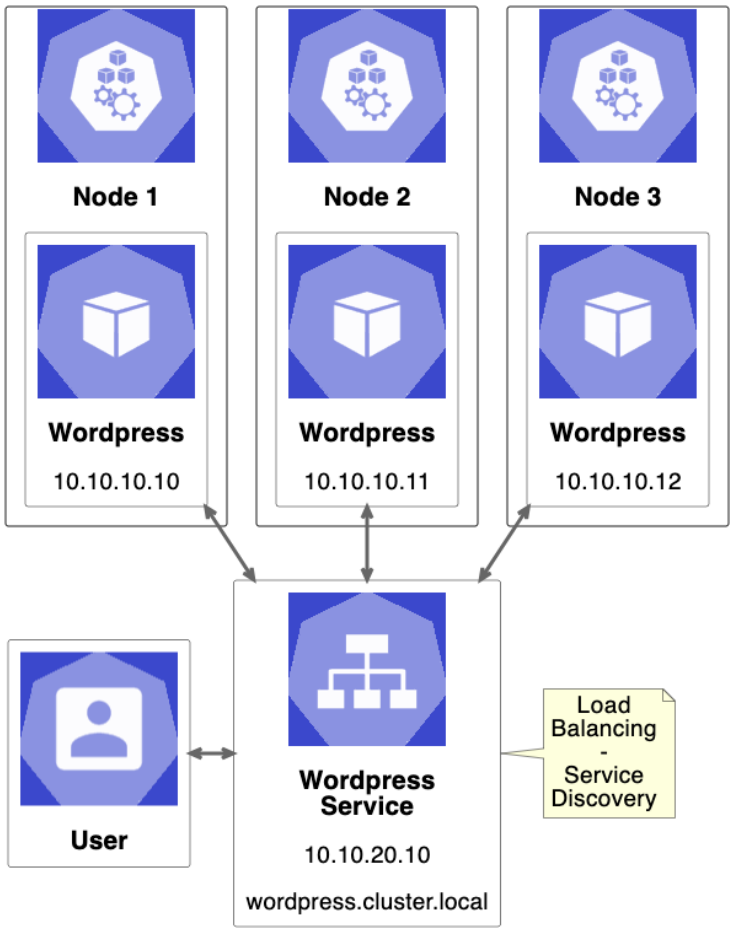
Services

Services are essentially reverse proxies that provide load balancing between Workloads.



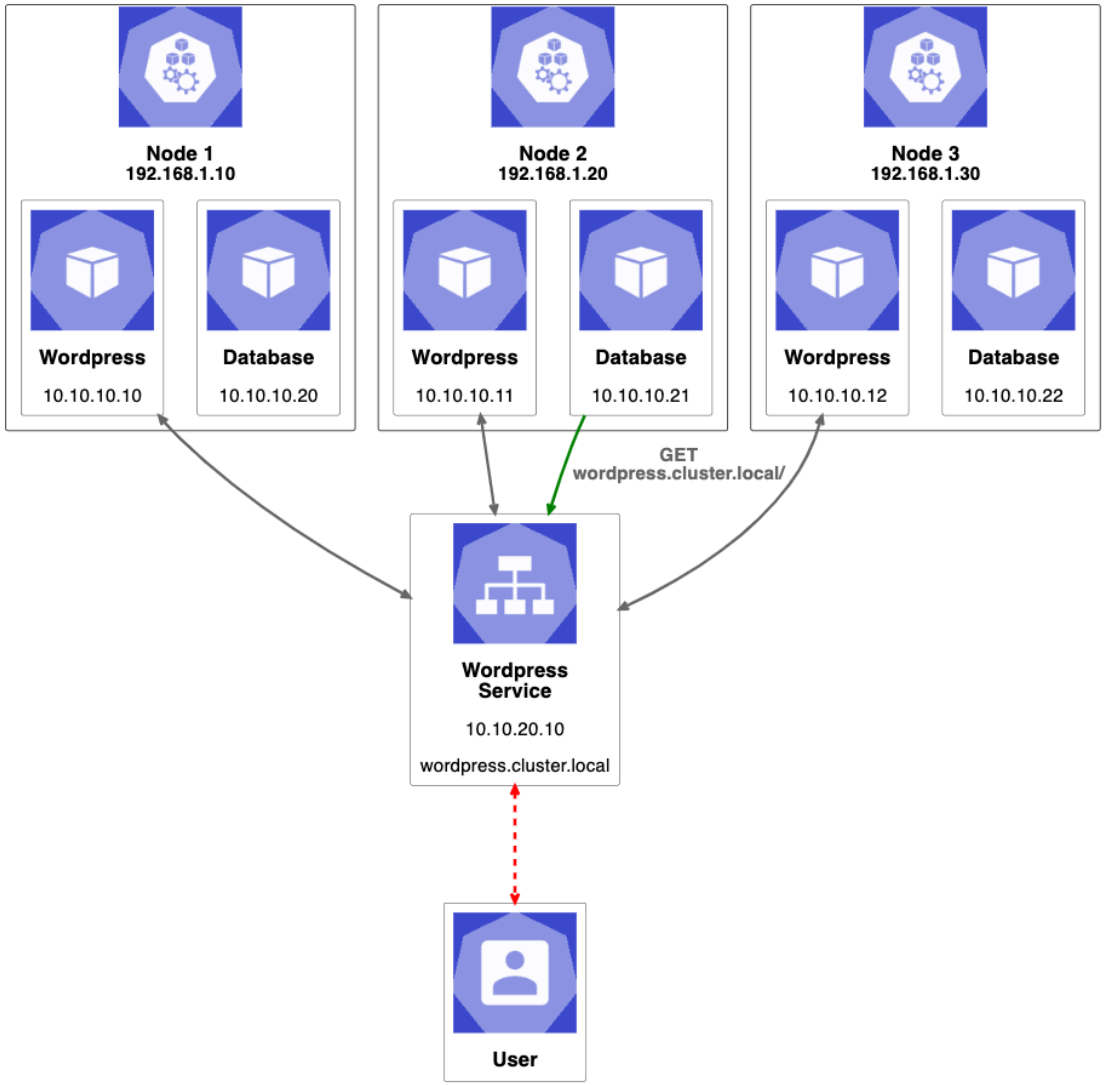
Services

Also provide a stable interface (IP) and/or domain name. Kubernetes also provides automatic load balancing between Pods. There are five types of Services.



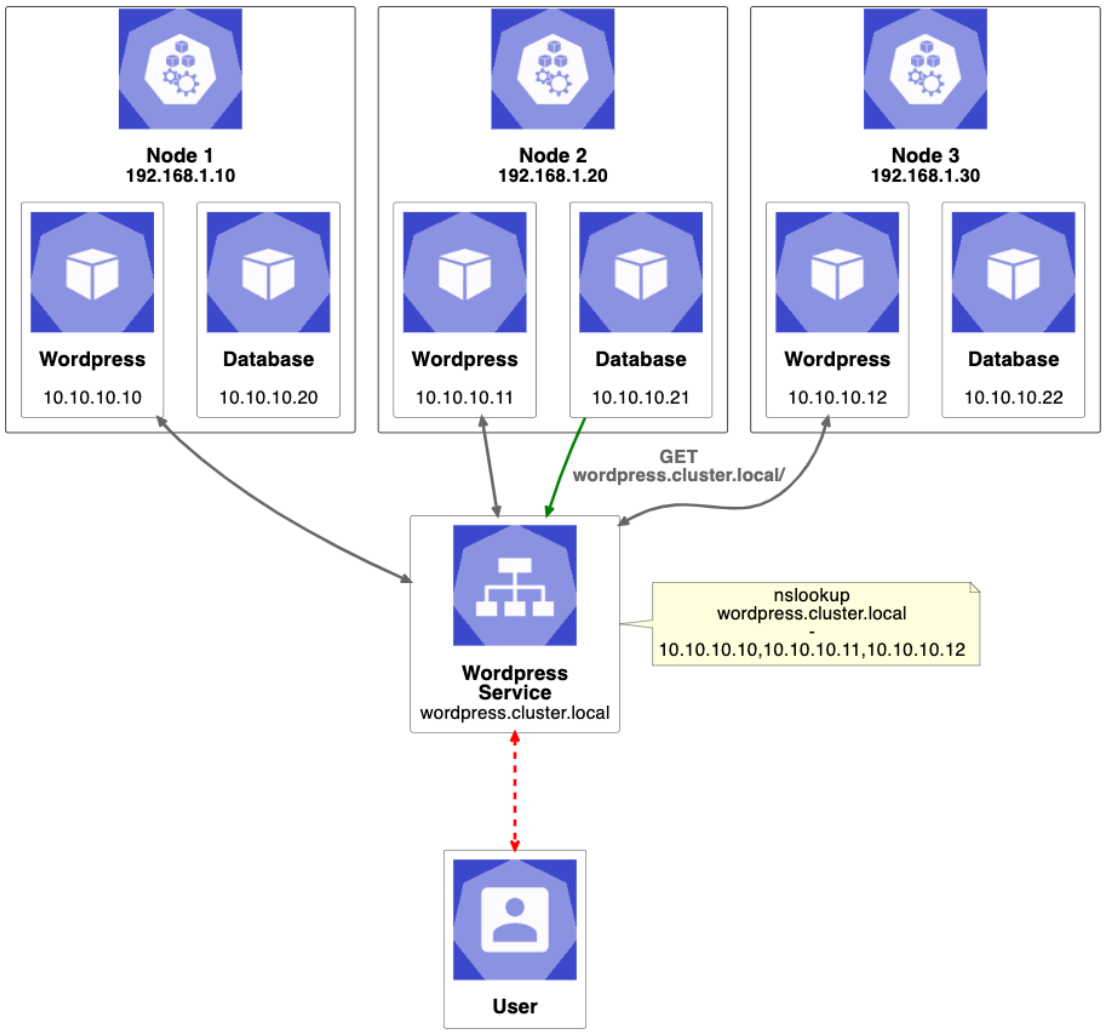
Service - Cluster IP

Cluster IP is accessible only from within the cluster.



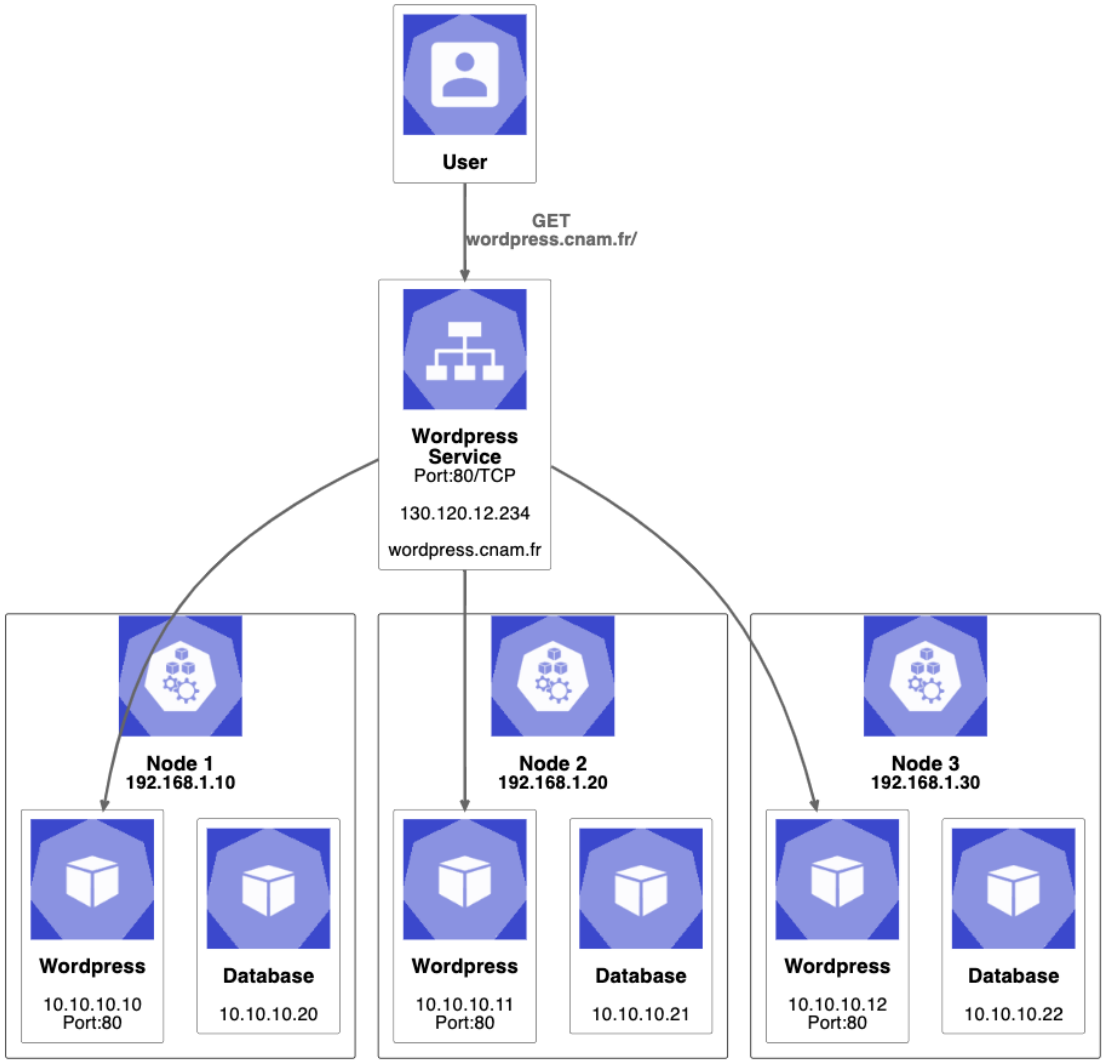
Service - Headless

Headless services do not have a cluster IP and are used for stateful applications.



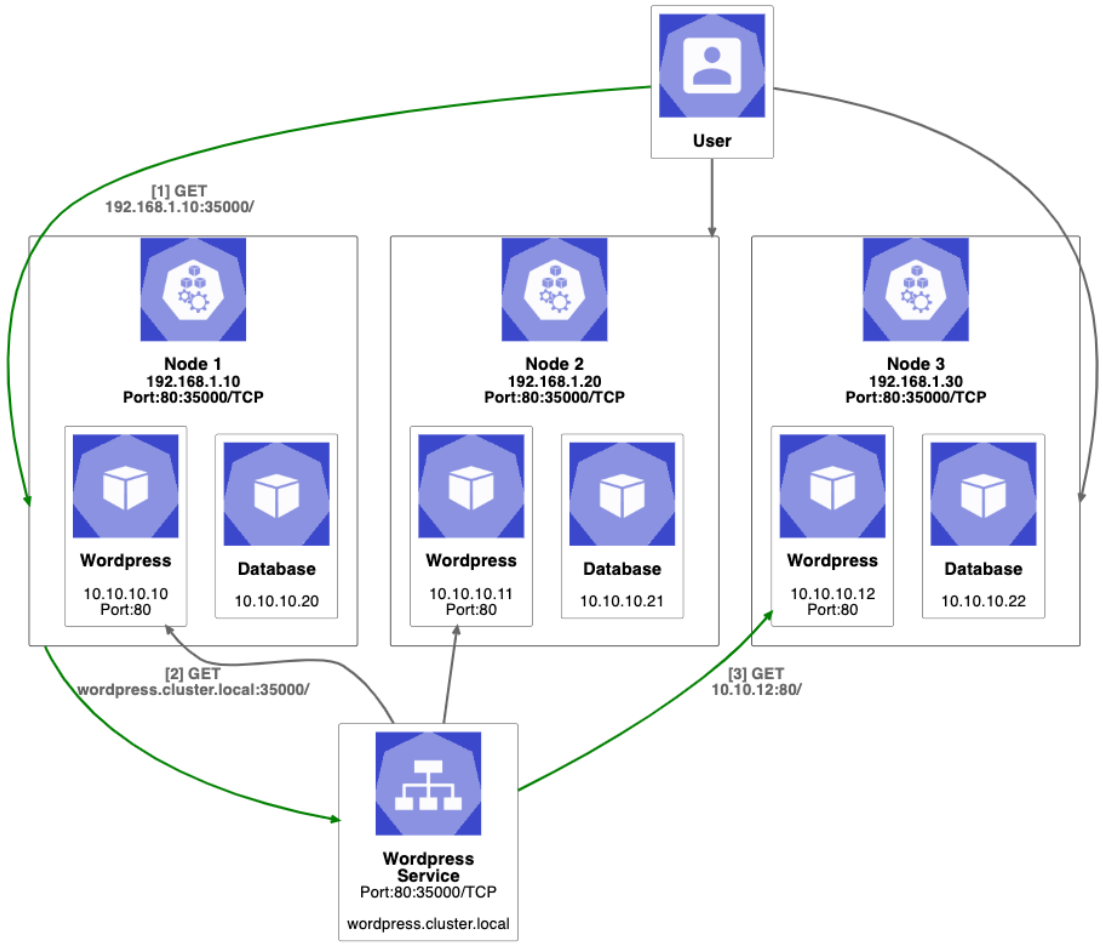
Service - LoadBalancer

LoadBalancer services are integrated with cloud providers to handle external traffic.



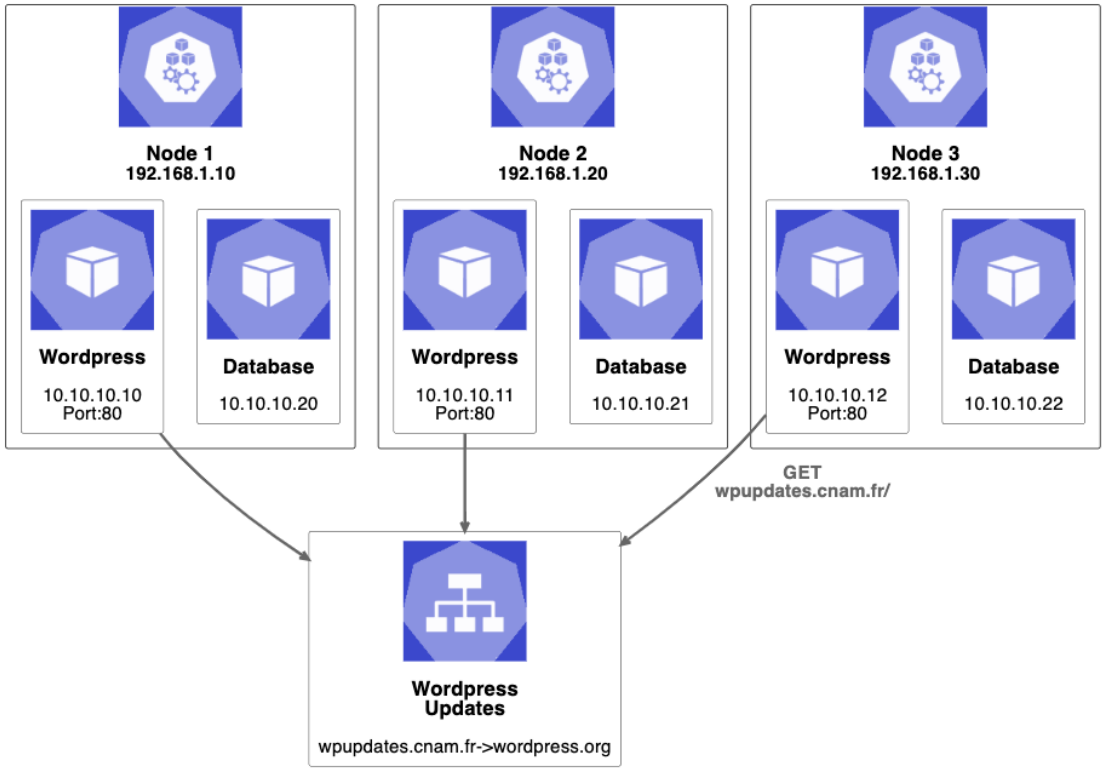
Service - NodePort

NodePort exposes the service on each node's IP on a static port.



Service - ExternalName

ExternalName maps the service to an external DNS name.



Labels and Selectors

If Pods are ephemeral, how can we group them?

Resources have labels to identify them. Selectors are used to group resources based on their labels.

- **Labels:** Key-value pairs attached to objects
- **Selectors:** Filter objects based on their labels

Example

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
  labels:
    app: frontend
...
---
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  selector:
    app: frontend
...
```

Network Policy resource

Network Policies

Restrict connections from/to Pods, Services and the Internet. By default, all traffic is allowed but if a Network Policy is defined, only the allowed traffic is permitted.

Target selection

- ➜ Labels
- ➜ Namespaces
- ➜ IP blocks

Rules content

- ➜ Target: Pods it applies to
- ➜ Type: Ingress or Egress
- ➜ From / To: Peer selector
- ➜ Protocol and port to allow

Namespaces

Some resources in Kubernetes are divided in Namespaces

Important distinction

Kubernetes Namespaces $\neq$ Linux Namespaces

Linux Namespaces:

- ➜ OS kernel feature
- ➜ Process-level isolation
- ➜ Isolate PID, network, mounts

Kubernetes Namespaces:

- ➜ API-level construct
- ➜ Resource organization
- ➜ Logical distinction of resources

Kubernetes Namespaces: Example

Example

A company may have separate namespaces for:

- ➜ GitLab deployment
- ➜ VPN service
- ➜ Monitoring applications

Namespaced resources include: Services, Deployments, ConfigMaps, NetworkPolicies

Default behavior

If you do not specify the namespace, resources are created in `default`

Configuration Management

Applications need configuration and secrets. How to inject them ?

Kubernetes provides two resources

- ➜ **ConfigMaps**: Non-sensitive configuration (URLs, feature flags, config files)
- ➜ **Secrets**: Sensitive data (passwords, API keys, certificates, tokens)

Separation of concerns

Configuration is managed separately from container images, enabling the same image to run in different environments (dev, staging, prod).

ConfigMaps

Create ConfigMap from literal values:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  database_url: "postgres://db-server:5432/mydb"
  log_level: "info"
  feature_flag: "true"
  app.properties: |
    server.port=8080
    max.connections=100
```

Use in Pod as environment variables:

```
env:
  - name: DATABASE_URL
    valueFrom:
      configMapKeyRef:
        name: app-config
        key: database_url
```

Secrets

Create Secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: db-credentials
type: Opaque
data:
  username: YWRtaW4=        # base64 encoded
  password: cGFzc3dvcmQ=    # base64 encoded
```

Use in Pod:

```
env:
  - name: DB_USERNAME
    valueFrom:
      secretKeyRef:
        name: db-credentials
        key: username
  - name: DB_PASSWORD
    valueFrom:
      secretKeyRef:
        name: db-credentials
        key: password
```

ConfigMaps vs Secrets: Best Practices

ConfigMaps:

- ➜ Plain text configuration
- ➜ Can be viewed by anyone with access
- ➜ Changed without restart (with proper setup)
- ➜ Version controlled safely

Secrets:

- ➜ Base64 encoded (not encrypted!)
- ➜ Restricted RBAC access
- ➜ Enable encryption at rest
- ➜ Never commit to version control
- ➜ Consider external secret managers

Security warning

Kubernetes Secrets provide basic protection but aren't secure by default.