Kubernetes Resources & Networking

hytsao (2026, CC)

Outline

- Kubernetes Object Model & kubectl
- Workload Resources
  - Pod, ReplicaSet, Deployment, DaemonSet, Job, CronJob
- Configuration & Storage
  - ConfigMap, Secret, Volume, PersistentVolume / PersistentVolumeClaim / StorageClass, StatefulSet
- Networking
  - Network Model, Service, DNS / CoreDNS, Ingress
- Health Checks
- Resource Management
  - Requests & Limits, QoS, LimitRange / ResourceQuota

Kubernetes Object Model

Kubernetes Object Model

- Everything in Kubernetes is represented as an Object stored in etcd
- Most Kubernetes manifests you write primarily include...

apiVersion

K8s provides an API that supports this object

kind

Object we are trying to create

metadata

The information about the object. e.g., name, label

spec

Defines what's inside the object we are creating

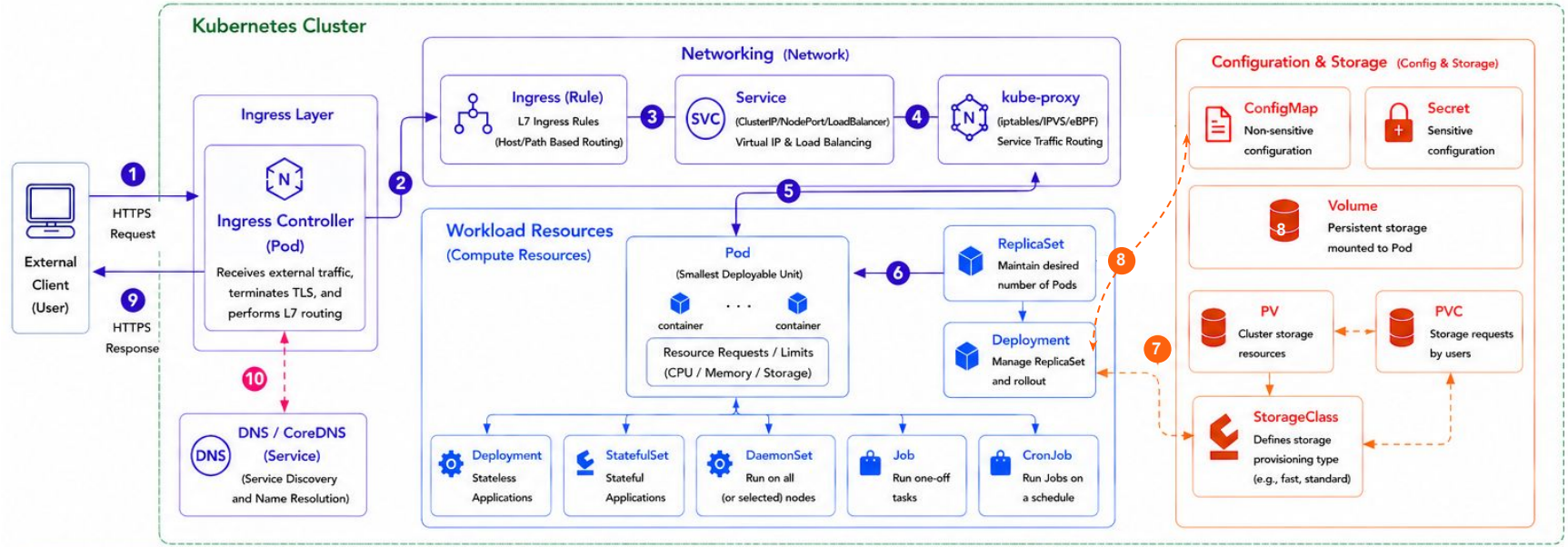
Pod

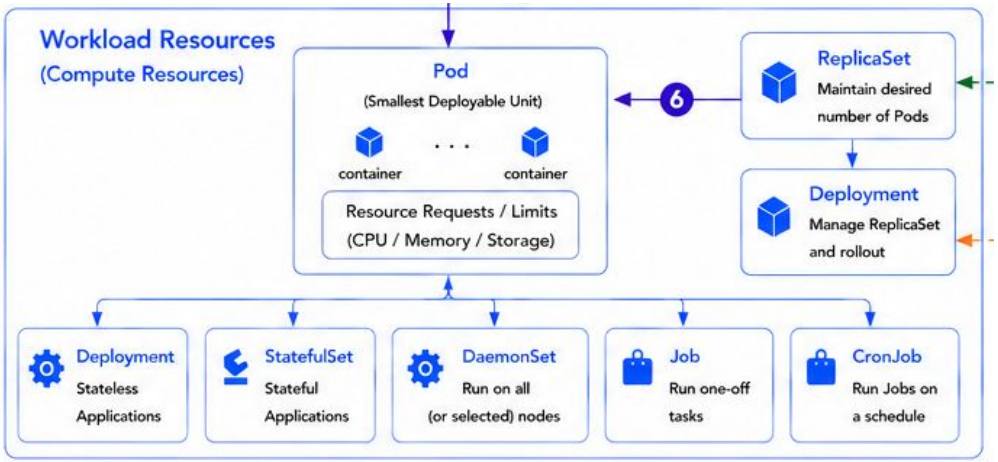
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-demo
  labels:
    app: web
spec:
  containers:
  - name: nginx
    image: "nginx:1.14.2"
    ports:
    - containerPort: 80
```

kubectl commands

- **Syntax:** `kubectl [command] [TYPE] [NAME] [flags]`
  - Apply / Create / Delete
    - `kubectl apply -f manifest.yaml` # create or update (idempotent)
    - `kubectl delete -f manifest.yaml` # remove
    - `kubectl delete pod my-pod` # remove by name
  - Inspect
    - `kubectl get pods -n <ns> -o wide` # list with node & IP
    - `kubectl describe pod <name>` # events, conditions, probes
    - `kubectl get events --sort-by=.lastTimestamp`

Overview


Kubernetes Cluster
Networking (Network)
Ingress (Rule)
L7 Ingress Rules
(Host/Path Based Routing)
Service
(ClusterIP/NodePort/LoadBalancer)
Virtual IP & Load Balancing
kube-proxy
(iptables/IPVS/eBPF)
Service Traffic Routing
Ingress Layer
Ingress Controller
(Pod)
Receives external traffic, terminates TLS, and performs L7 routing
External Client (User)
HTTPS Request
HTTPS Response
Workload Resources
(Compute Resources)
Pod
(Smallest Deployable Unit)
container
container
Resource Requests / Limits
(CPU / Memory / Storage)
ReplicaSet
Maintain desired number of Pods
Deployment
Manage ReplicaSet and rollout
Deployment
Stateless Applications
StatefulSet
Stateful Applications
DaemonSet
Run on all (or selected) nodes
Job
Run one-off tasks
CronJob
Run Jobs on a schedule
DNS / CoreDNS
(Service)
(Service Discovery and Name Resolution)
Configuration & Storage (Config & Storage)
ConfigMap
Non-sensitive configuration
Secret
Sensitive configuration
Volume
Persistent storage mounted to Pod
PV
Cluster storage resources
PVC
Storage requests by users
StorageClass
Defines storage provisioning type (e.g., fast, standard)



Workload Resources

Including: Pod, ReplicaSet, Deployment, DaemonSet, Job, CronJob

Pod

- Pod is the smallest deployable unit in Kubernetes
  - A Pod wraps one or more containers that share resources
- Shared resources inside a Pod
  - Network namespace: same IP address and port space
    - Containers communicate via localhost
  - Storage volumes: containers can read/write the same Volumes
  - Lifecycle: created, restarted, and terminated together
- Key properties
  - Each Pod gets a unique cluster-internal IP (flat network)
  - Pod IPs are ephemeral – they may change when the Pod is recreated
  - Pod spec is mostly immutable once created (delete → recreate to change)

Pod - multiple containers

- Multi-container patterns
  - Sidecar - augments main container (log forwarder, proxy, sync)
  - Init container - runs before main containers, prepares environment
  - Ephemeral - injected temporarily for debugging (`kubectl debug`)
- Pods that run multiple containers that need to work together
- Best practice is to run only one container per pod

Pod - phases

- Pod phases (`kubectl get pod → STATUS` column)
  - Pending - accepted by cluster; waiting for scheduling or image pulling
  - Running - at least one container is running
  - Succeeded - all containers exited 0 (Jobs use this)
  - Failed - all containers terminated; at least one non-zero exit
  - Unknown - node is unreachable
- Container states (inside a running Pod)
  - Waiting – not running yet (image pull, init containers pending)
  - Running – executing normally
  - Terminated – exited; check exitCode and reason field

Pod - restartPolicy and conditions

- restartPolicy – controls container restart behavior
  - Always – always restart on exit (default for long-running apps)
  - OnFailure – restart only on non-zero exit
  - Never – never restart
- Pod conditions (inspect with kubectl describe)
  - PodScheduled
  - ContainersReady
  - Initialized
  - Ready
    - Pod healthy (all Readiness probes passing)
    - Only Ready Pods receive traffic

Pod - YAML Manifest

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: hello
  namespace: testing
  labels:
    app: hello
spec:
  containers:
    - name: hello
      image: 'busybox:1.28'
      command:
        - sh
        - '-c'
        - 'echo "Hello, Kubernetes!" && sleep 3600'
      ports:
        - containerPort: 80
  restartPolicy: OnFailure
```

```
# kubectl describe pod hello -n testing

Name:             hello
Namespace:        testing
Node:             na-k8s-w2/172.16.1.6
Labels:           app=hello
Annotations:      cni.projectcalico.org/podIP: 10.244.24.207/32
Status:           Running
IP:               10.244.24.207
Containers:
  hello:
    Container ID:  cri-o://c8a52226220e712d8a929566f60b8529a50a79c488c01a6499f1d54f3c3827a6
    Image:         busybox:1.28
    Image ID:      docker.io/library/busybox@sha256:141c253bc4c3fd0a2
    Port:          80/TCP
    Command:
      sh
      -c
      echo "Hello, Kubernetes!" && sleep 3600
    State:          Running
    Ready:          True
Conditions:
  Type                        Status
  Initialized                 True
  Ready                       True
  ContainersReady             True
  PodScheduled                True
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  21s   default-scheduler  Successfully assigned testing/hello to na-k8s-w2
  Normal  Pulling    21s   kubelet            spec.containers{hello}: Pulling image "busybox:1.28"
  Normal  Pulled     15s   kubelet            spec.containers{hello}: Successfully pulled image
  Normal  Created    15s   kubelet            spec.containers{hello}: Created container: hello
  Normal  Started    15s   kubelet            spec.containers{hello}: Started container hello
```

Labels, Annotations & Selectors

- **Labels**: key/value pairs attached to objects
  - Used by selectors to group and identify objects.
    - For example: `app=nginx`
- **Annotations**: arbitrary metadata NOT used for selection
  - Larger values allowed (URLs, JSON, etc.)
  - Used by tools: monitoring hints, CI/CD commit SHA, timestamp, etc.
- **Selectors**: query mechanism based on labels
  - Equality. For example: `app=nginx / app!=redis`
  - Set-based. For example: `app in (nginx, redis) / env notin (dev, test)`

➢ If something needs to be queried → use a **Label**
➢ If it is informational only → use an **Annotation**

ReplicaSet (rs)

- Ensures a specified number of identical Pod replicas are always running
  - Creates Pods when count falls below desired; removes extras
  - Uses a label selector to identify "owned" Pods
- How selectors work
  - `spec.selector.matchLabels` MUST match `spec.template.metadata.labels`
  - K8s can also adopt existing Pods that match the selector
- Limitation: no built-in rolling update
- Use Deployment instead – it manages ReplicaSets for you
  - Deployment adds rolling update, rollback, and revision history

ReplicaSet - YAML Manifest

- replicas: desired count
- selector must exactly match template labels
- template is a Pod spec without apiVersion and kind
- Scaling:
  - `kubectl scale rs/nginx-rs --replicas=5`

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-rs
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: 'nginx:1.25'
          ports:
            - containerPort: 80
          resources:
            requests:
              cpu: 100m
              memory: 64Mi
            limits:
              cpu: 500m
              memory: 128Mi
```

Deployment (deploy)

- Manages ReplicaSets and adds declarative rolling update / rollback
  - When you update the Pod spec, Deployment creates a new ReplicaSet
  - Gradually scales up new ReplicaSet while scaling down old ReplicaSet
- Update strategies
  - RollingUpdate (default) – zero-downtime upgrade
    - maxUnavailable: max Pods that can be down during update (default 25%)
    - maxSurge: max extra Pods above desired count (default 25%)
  - Recreate – kill **ALL** old Pods first, then create new ones (causes downtime)
- minReadySeconds: seconds new Pod must be Ready before counted available

Deployment (deploy)

- Rollback
  - K8s keeps revision history (revisionHistoryLimit, default 10)
  - `kubectl rollout history deployment/<name>`
  - `kubectl rollout undo deployment/<name>` → previous revision
  - `kubectl rollout undo deployment/<name> --to-revision=2`
- Best suited for **stateless** applications
  - Pods are interchangeable; no stable identity required
  - Typical: web servers, REST APIs, background workers

Deployment - YAML Manifest

- replicas: 3
- revisionHistoryLimit: 5
- minReadySeconds: 10
- selector must exactly match template labels
- strategy.rollingUpdate
  - maxUnavailable: 1
  - maxSurge: 1
  - minReadySeconds: 10
    - wait 10s after Ready before proceeding

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deploy
  labels:
    app: nginx
spec:
  replicas: 3
  revisionHistoryLimit: 5
  minReadySeconds: 10
  selector:
    matchLabels:
      app: nginx
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: 'nginx:1.25'
          ports:
            - containerPort: 80
          resources:
            requests:
              cpu: 100m
              memory: 64Mi
            limits:
              cpu: 500m
              memory: 128Mi
```

DaemonSet (ds)

- Ensures exactly ONE Pod runs on every (or selected) Node
  - Node joins cluster → DaemonSet Pod automatically scheduled on it
  - Node removed → its DaemonSet Pod is garbage collected
- Typical use cases
  - Log collection: Fluentd, Filebeat, Promtail
  - Metrics collection: Prometheus Node Exporter, Datadog Agent
  - Network plugin: kube-proxy itself, CNI plugins (Calico, Cilium)
  - Storage driver: Ceph CSI, host volume management

DaemonSet (ds)

- No replicas field – count is determined by node count
- Update strategies
  - RollingUpdate (default) – replaces Pods one by one
  - OnDelete – replace only when Pod is manually deleted
- DaemonSet vs Deployment
  - Deployment: "I want N replicas anywhere"
  - DaemonSet: "I want one on EACH node"

[App.] DaemonSet - YAML Manifest

- No replicas field
- tolerations able to run on
  - control-plane or other tainted nodes

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: logging
  labels:
    k8s-app: fluentd
spec:
  selector:
    matchLabels:
      k8s-app: fluentd
  template:
    metadata:
      labels:
        k8s-app: fluentd
    spec:
      containers:
        - name: fluentd-elasticsearch
          image: 'quay.io/fluentd_elasticsearch/fluentd:v5.0.1'
          volumeMounts:
            - name: varlog
              mountPath: /var/log
      volumes:
        - name: varlog
          hostPath:
            path: /var/log
```

[App.] Job - One-Shot Tasks

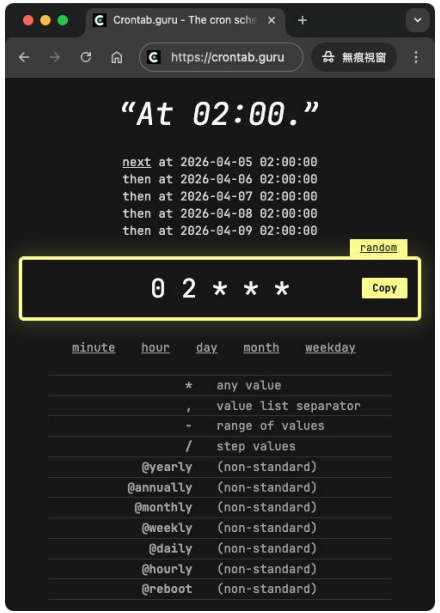
- Runs a Pod to completion – task finishes, Pod is not restarted
  - Unlike Deployment, completion is success; K8s tracks it
- Key spec fields
  - completions – how many successful completions needed (default 1)
  - parallelism – how many Pods can run simultaneously (default 1)
  - backoffLimit – max retries before Job is marked Failed (default 6)
  - activeDeadlineSeconds – hard wall-clock limit
  - ttlSecondsAfterFinished – auto-clean up Job after N seconds
- restartPolicy must be OnFailure or Never (cannnot be Always)

[App.] Job - One-Shot Tasks

- Completion modes
  - NonIndexed (default) – any Pod completion counts
  - Indexed – each Pod gets index 0…N-1, useful for work-queue sharding
- Use cases
  - Database schema migration before deploying new app version
  - Batch data processing (parallel workers across large dataset)
  - One-time report generation or file conversion

[App.] CronJob (cron) – Scheduled Tasks

- Creates a Job on a cron schedule
  - Cron syntax: "minute hour day month weekday"
    - `"0 2 * * *"` = every day at 02:00
    - `"*/5 * * * *"` = every 5 minutes
- Key spec fields
  - schedule – cron expression (UTC by default)
  - timeZone – e.g. "Asia/Taipei" (K8s 1.27+)
  - concurrencyPolicy – behavior when previous Job is still running
    - Allow (default) / Forbid / Replace
  - startingDeadlineSeconds – skip if Job cannot start within N seconds
  - successfulJobsHistoryLimit (default 3) / failedJobsHistoryLimit (default 1)



https://crontab.guru/

[App.] CronJob (cron) – Scheduled Tasks

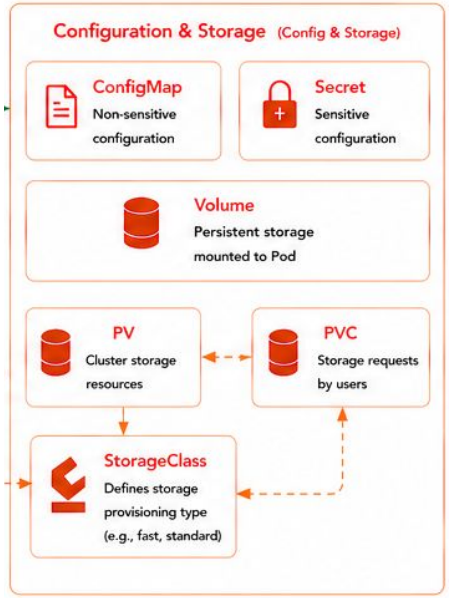
- Use cases
  - Database backup (nightly)
  - Cache warm-up / stale entry cleanup
  - Scheduled reports, notification emails
- Debug tip: trigger manually without waiting for the schedule
  - `kubectl create job test --from=cronjob/<name>`

[App.] Job & CronJob - YAML Manifest

- Job
  - restartPolicy: OnFailure
  - backoffLimit: 3
  - ttlSecondsAfterFinished auto cleanup
- CronJob
  - wraps a jobTemplate which is a Job spec
  - concurrencyPolicy:
    - Forbid prevents overlapping runs

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: db-migration
spec:
  completions: 1
  parallelism: 1
  backoffLimit: 3
  ttlSecondsAfterFinished: 600
  template:
    spec:
      restartPolicy: OnFailure
      containers:
        - name: migrate
          image: 'myapp:latest'
          command:
            - python
            - manage.py
            - migrate
```

```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: db-migration
spec:
  schedule: 0 2 * * *
  timeZone: Asia/Taipei
  concurrencyPolicy: Forbid
  successfulJobsHistoryLimit: 3
  failedJobsHistoryLimit: 1
  jobTemplate:
    spec:
      completions: 1
      parallelism: 1
      backoffLimit: 3
      ttlSecondsAfterFinished: 600
      template:
        spec:
          restartPolicy: OnFailure
          containers:
            - name: migrate
              image: 'myapp:latest'
              command:
                - python
                - manage.py
                - migrate
```



Configuration & Storage

Including: ConfigMap, Secret, Volume, PV / PVC / StorageClass

ConfigMap (cm) – Non-Sensitive data

- Stores non-sensitive config as key-value pairs
  - Decouples configuration from the container image
  - Can store simple values, multi-line text, or entire config files
- Size limit: 1 MiB per ConfigMap
- `kubectl create configmap my-cm --from-file=config.yaml`
- ConfigMap YAML Manifest
  - data:
    - key-value
    - | for multi-line

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_ENV: production
  LOG_LEVEL: info
  nginx.conf: |
    server {
      listen 80;
      location / { proxy_pass http://backend; }
    }
```

Secret – Sensitive Data

- Same methods as ConfigMap, but intended for passwords, tokens, TLS certificates, SSH keys
- Secret data is **NOT encrypted** by default
  - Values are base64-encoded (encoding ≠ encryption)
  - Anyone who can GET the Secret object sees the decoded data
  - Stored as base64 in etcd (plaintext unless etcd encryption enabled)
- Production hardening
  - Enable etcd encryption-at-rest
  - Use RBAC to restrict who can read Secrets (least privilege)
  - Consider external secret stores
    - HashiCorp Vault, AWS Secrets Manager, Azure Key Vault
    - External Secrets Operator, Sealed Secrets

Secret – secret types

- Common Secret types
  - Opaque – generic, user-defined key-value pairs
  - kubernetes.io/tls – TLS certificate (tls.crt) + key (tls.key)
  - kubernetes.io/dockerconfigjson – image pull credentials
- Secret YAML Manifest
  - stringData:
    - auto base64-encodes
  - data:
    - needs pre-encoded values

```yaml
# format 1
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
stringData:
  DB_USER: admin
  DB_PASSWORD: s3cr3t!
```

```yaml
# format 2
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
data:
  DB_USER: YWRtaW4=
  DB_PASSWORD: czNjcjN0IQ==
```

Volumes

- The mechanism Kubernetes uses to solve data storage and data sharing
- Why volumes
  - Data persistence: Container filesystem is ephemeral (wiped on container restart)
  - Shared storage: Multiple containers in a Pod need to share data
- Volumes type, according to "Life expectancy"
  - Ephemeral Volumes
    - Data deleted when Pod is deleted
    - Use case: caching services
  - Persistent Volumes
    - Data persists even if the Pod is deleted
    - Use case: database, logs

Volumes

- Volumes type, according to "Implementation approach"
  - emptyDir – Temporary storage created on Pod startup
    - Tied to Pod lifetime
    - Data persists across container restarts but deleted on Pod removal
    - Storage: Local disk (default) or RAM (medium: Memory)
    - Use Cases: Scratch space (merge sort), shared buffer for sidecars, caching
  - hostPath – Mounts a file/directory from the Host Node's filesystem
    - Breaks Portability (Node-dependent) and poses Security risks
    - Use Cases: DaemonSets needing system logs (/var/log) or node internals

Volumes

- Volumes type, according to "Implementation approach"
  - configMap / secret – Injects cluster data as read-only files
    - "Projects" keys from etcd into the container filesystem
    - Hot-sync: Files auto-update (~1 min) when source changes
      - Except for `subPath` mounts
    - Security: Secrets use tmpfs (RAM-backed) to avoid writing to Node disk
  - downwardAPI – Makes Pod/Container metadata available to processes
    - Exposes Pod and Container metadata as files
      - e.g., metadata.name, status.podIP, requests.cpu
    - Benefit: Enables "environment-aware" apps without an API client

Volumes

- Volumes type, according to "Implementation approach"
  - NFS – Network File System mount
    - Support: ReadWriteMany (multiple Nodes can access simultaneously)
    - Persistence: Data remains even after the Pod is deleted
    - Use case: legacy app data sharing, centralized file storage
  - CSI (Container Storage Interface) – Modern standard for external storage
    - Nature: Replaces legacy "In-tree" drivers
    - Capabilities: Volume snapshots, resizing, and vendor-specific storage (AWS, GCP, Azure)

[App.] Configure a Pod to Use a Configuration files

- Two ways to consume a Configuration in a Pod
  - As environment variables
    - Inject with
      - `envFrom.[configMapRef|secretRef]` - inject ALL keys as env vars
      - `env[].valueFrom.[configMapKeyRef|secretKeyRef]` - inject ONE specific key (recommended)
    - Command-line arguments via `$(ENV_VAR)` substitution
  - Volume mount with files
    - Each key becomes a file in the mounted directory
    - Best for config files: nginx.conf

[App.] Pod YAML Manifest with Ephemeral Volumes

- As environment variables
- Volume mount with files
  - mount one key as a specific file

```yaml
# format 1
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
stringData:
  DB_USER: admin
  DB_PASSWORD: s3cr3t!
```

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_ENV: production
  LOG_LEVEL: info
  nginx.conf: |
    server {
      listen 80;
      location / { proxy_pass http://backend; }
    }
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: cm-demo-pod
spec:
  containers:
    - name: app
      image: 'myapp:latest'
      command:
        - /bin/sh
        - '-c'
      args:
        - >-
          echo 'Starting in $(APP_ENV) mode with log level $(MY_LOG_LEVEL)';
          /entrypoint.sh
      env:
        - name: MY_LOG_LEVEL
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: LOG_LEVEL
      envFrom:
        - configMapRef:
            name: app-config
        - secretRef:
            name: db-secret
      volumeMounts:
        - name: cfg-vol
          mountPath: /etc/nginx/nginx.conf
          subPath: nginx.conf
  volumes:
    - name: cfg-vol
      configMap:
        name: app-config
```

Persistent volumes

- Persistent volumes
  - Data persists even if the Pod is deleted
  - Decoupling of storage supply and demand
    - PersistentVolume (PV) – Supply a "Resource" in the cluster
    - PersistentVolumeClaim (PVC) – Demand a "Request" for storage

PersistentVolume (PV)

- **PersistentVolume (PV) – Supply a "Resource" in the cluster**
  - A piece of storage in the cluster that has been provisioned
  - Scope: Cluster-wide (not namespaced)
  - Its lifecycle is independent of any individual Pod that uses the PV
  - Captures the details of the implementation, be it NFS, iSCSI, or a cloud-provider-specific storage system (via CSI)
- Provisioning Types:
  - Static Provisioning: A cluster administrator creates several PVs manually
  - Dynamic Provisioning: Via StorageClasses (defined below)

[App.] PersistentVolume - YAML Manifest

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-demo
spec:
  capacity:
    storage: 10Gi
  accessModes:
  - ReadWriteOnce
  reclaimPolicy: Retain
  storageClassName: standard
  hostPath:
    path: /mnt/data
```

PersistentVolumeClaim (PVC)

- **PersistentVolumeClaim (PVC) – Demand a "Request" for storage**
  - Pods use PVCs as volumes. The cluster inspects the claim to find the bound volume and mounts that volume for the Pod (Binding)
  - Allows a developer to request specific size and access modes (e.g., ReadWriteOnce, ReadOnlyMany, ReadWriteMany) without knowing the underlying storage provider
  - Scope: Namespaced (exists within a Namespace)

[App.] PersistentVolumeClaim - YAML Manifest

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pod-demo-pvc
spec:
  storageClassName: standard
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

StorageClass (SC)

- **StorageClass – The "Blueprint" for dynamic provisioning**
  - Allows admin to describe the "classes" of storage they offer
    - e.g., "fast-ssd" vs "standard-hdd"
  - Dynamic Provisioning
    - Enables storage volumes to be created on-demand
    - When a PVC requests a specific StorageClass, the cluster automatically provisions the backend storage and creates the PV

[App.] StorageClass - YAML Manifest

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp3
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
allowVolumeExpansion: true
```

[App.] Persistent volumes supplement

- Access Modes (can specify one or more access modes)
  - ReadWriteOnce (RWO): Mounted as read-write by a single node.
  - ReadOnlyMany (ROX): Mounted read-only by many nodes.
  - ReadWriteMany (RWX): Mounted as read-write by many nodes (common for NFS).
- Reclaim Policy
  - Retain: Keeps the data after the PVC is deleted, allowing for manual recovery
  - Delete: Automatically removes the PV and the associated storage asset from the external infrastructure (e.g., AWS EBS or GCP PD)
  - Recycle: Performs a basic scrub (rm -rf /thevolume/*) to make the volume available again (deprecated in favor of dynamic provisioning)

[App.] StatefulSet (sts)

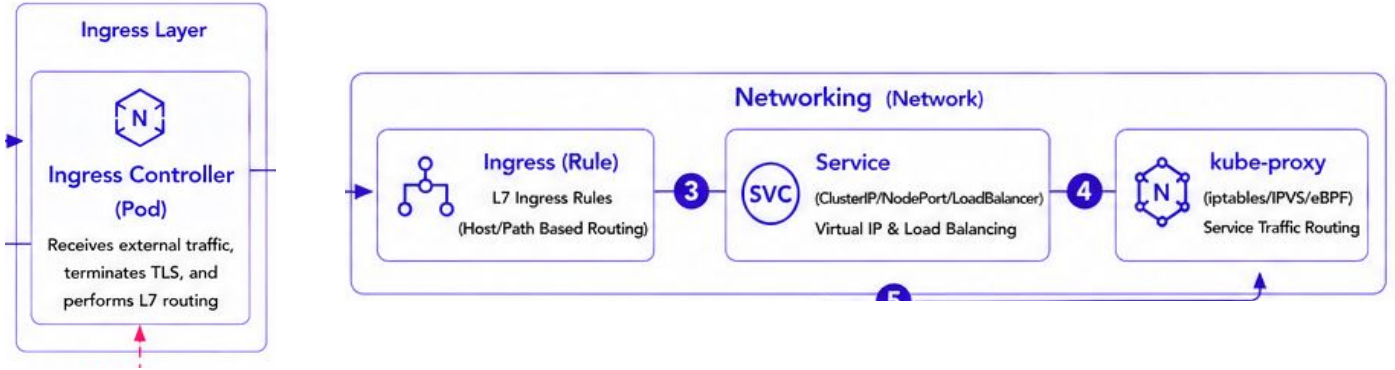
- Deployment Pods are interchangeable – no stable identity
- Stateful apps need guaranteed identity across rescheduling
  - Databases (MySQL, Postgres), message brokers (Kafka), caches (Redis Cluster)
- StatefulSet provides three guarantees
  - Stable Pod name: `app-0, app-1, app-2` → survive restarts
  - Stable DNS per Pod: `app-0.<headless-svc>.<namespace>.svc.cluster.local`
  - Stable storage: each Pod binds to its own dedicated PersistentVolumeClaim (PVC)
    - PVC survives Pod deletion → data is NOT lost on reschedule
- Ordered operations
  - Scale up: `app-0` must be Ready before `app-1` is created
  - Scale down: `app-2` is deleted first, then `app-1`, then `app-0`
  - Safe for leader election and replication bootstrapping

[App.] StatefulSet - YAML

- serviceName → links to the Headless Service
- Headless Service
  - clusterIP: None
  - enables per-Pod DNS
- volumeClaimTemplates
  - auto-creates PVCs

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mysql
spec:
  clusterIP: None
  selector:
    app: mysql
  ports:
    - port: 3306
```

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  serviceName: mysql
  replicas: 3
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
        - name: mysql
          image: 'mysql:8.0'
          volumeMounts:
            - name: data
              mountPath: /var/lib/mysql
  volumeClaimTemplates:
    - metadata:
        name: data
      spec:
        accessModes:
          - ReadWriteOnce
        storageClassName: my-storage-class
        resources:
          requests:
            storage: 1Gi
```



Networking

Including: Network Model, Service, DNS / CoreDNS, Ingress, NetworkPolicy

Kubernetes Network Model

- K8s expects a flat, routable Pod network model, which CNIs implement
  - Every Pod gets a unique cluster-wide IP address
  - Any Pod can communicate with any other Pod without NAT
  - Any Node can communicate with any Pod without NAT
- Implemented by CNI (Container Network Interface) plugins
  - Flannel – simple VXLAN overlay; does NOT support NetworkPolicy
  - Calico – BGP or VXLAN; supports NetworkPolicy; high performance
  - Cilium – eBPF-based; advanced observability, L7 policy, no kube-proxy
  - Weave Net – encrypted overlay mesh

Kubernetes Network Model

- Three distinct network planes
  - Node network – physical/VM network Nodes communicate on
  - Pod network – virtual overlay for Pods (e.g. 10.42.0.0/16)
  - Service network – virtual IP range for Services (e.g. 10.43.0.0/16)
- Pod IPs are ephemeral – they may change when the Pod is recreated
  - You should never hardcode Pod IPs
  - This is exactly why the Service abstraction exists

Service (svc) – The Stable Entry Point

- Service provides a stable IP + DNS name for a dynamic group of Pods
  - Pods come and go (ephemeral IPs); Service IP never changes
  - Uses **label selectors** to dynamically identify healthy target Pods
- Internal mechanics – how traffic actually flows
  - Create Service object → K8s assigns a ClusterIP (virtual IP)
  - EndpointSlice controller watches Pods/Services and maintains EndpointSlice objects
    - EndpointSlices represent backend endpoints for a Service, including endpoint addresses and ports
  - kube-proxy on every Node reads Service + EndpointSlices
  - kube-proxy writes nftables / iptables rules on each Node
  - Traffic → ClusterIP → DNAT (iptables) → real Pod IP

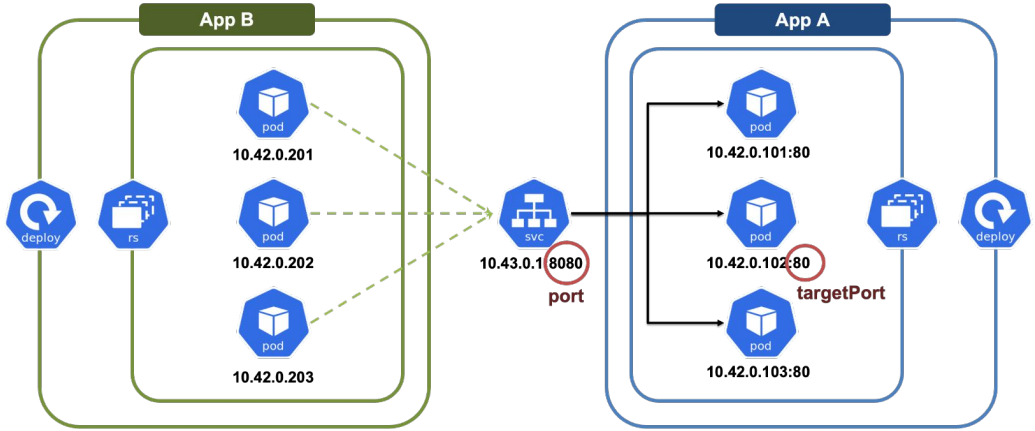
Service (svc)

- ClusterIP is a virtual IP – does not exist on any NIC
  - Lives only in iptables/ipvs rules in the kernel
  - That is why ping ClusterIP usually yields no response
- Service Types
  - ClusterIP (default)
  - NodePort
  - LoadBalancer
  - ExternalName
  - Headless (`clusterIP: None`)

Service (svc) - ClusterIP

- The default type of service
- Allocates a stable virtual IP reachable only inside the cluster
- DNS: `<serviceName>.<namespace>.svc.cluster.local` → ClusterIP
- Use for: inter-service communication, internal APIs

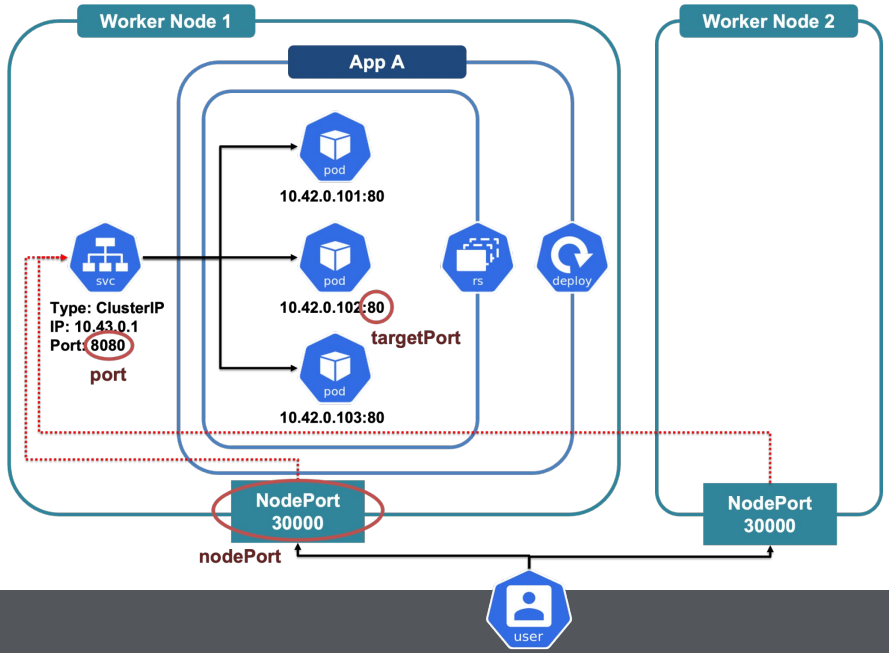
```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-svc
spec:
  type: ClusterIP
  selector:
    app: nginx
  ports:
  - port: 8080     # Service port (stable)
    targetPort: 80 # Pod port
```



Service (svc) - NodePort

- Exposes Service on a fixed port (30000-32767) on EVERY Node
- External access: `<any-NodeIP>:<nodePort>`
- K8s auto-routes to ClusterIP → Pod
- Use for: dev/test, on-prem without cloud LB

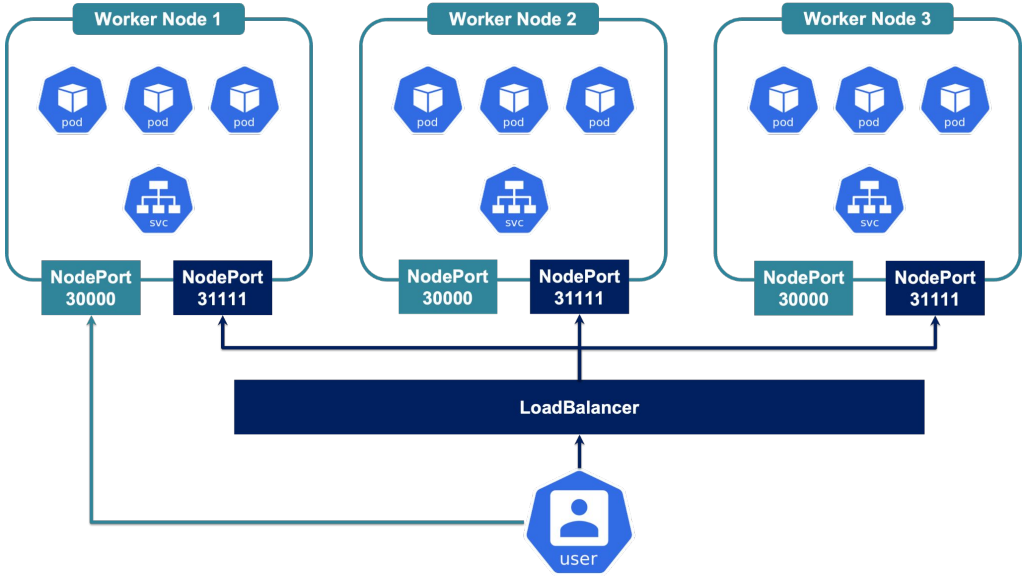
```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-nodeport
spec:
  type: NodePort
  selector:
    app: nginx
  ports:
  - port: 8080       # Service port (stable)
    targetPort: 80   # Pod port
    nodePort: 30080 # Node port fixed or omit
```



Service (svc) - LoadBalancer

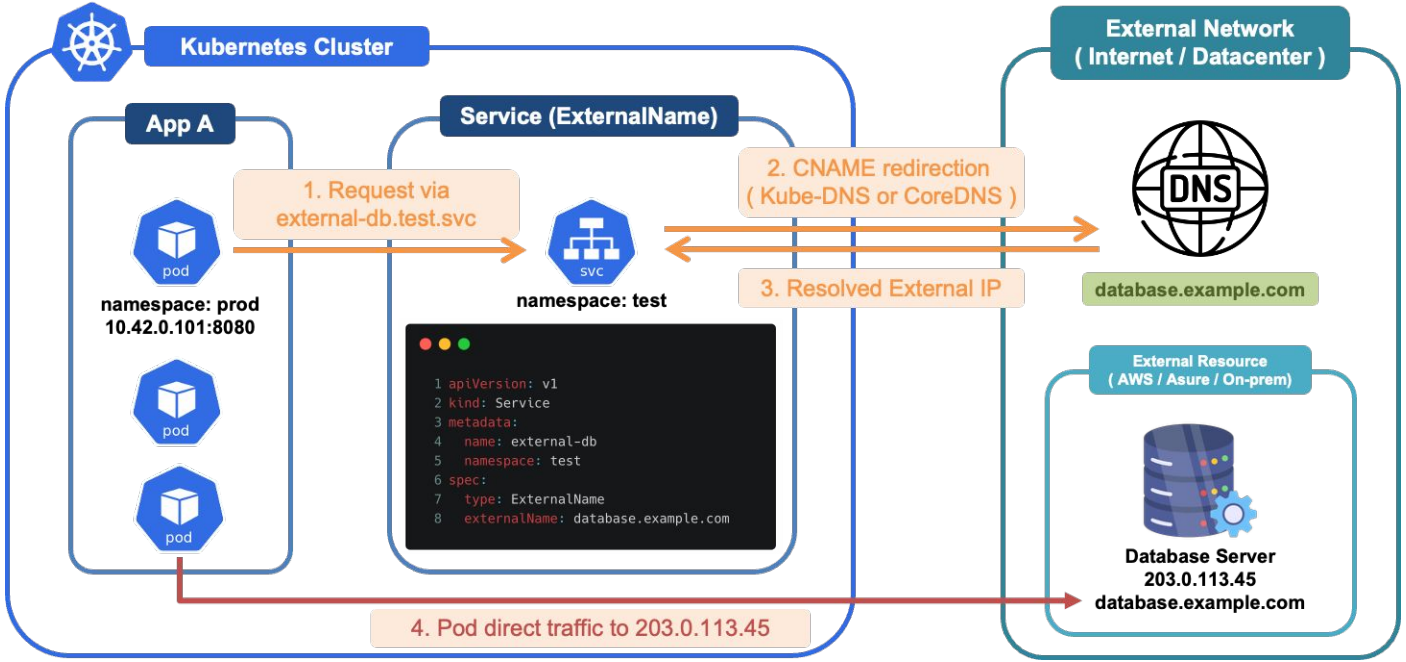
- Extends NodePort; additionally provisions a cloud load balancer
- Cloud controller creates a public IP / DNS and routes traffic to NodePort
- On-premise: use MetalLB to emulate cloud LB behavior

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-loadbalancer
spec:
  type: LoadBalancer
  selector:
    app: nginx
  ports:
  - port: 8080      # Service port (stable)
    targetPort: 80  # Pod port
```



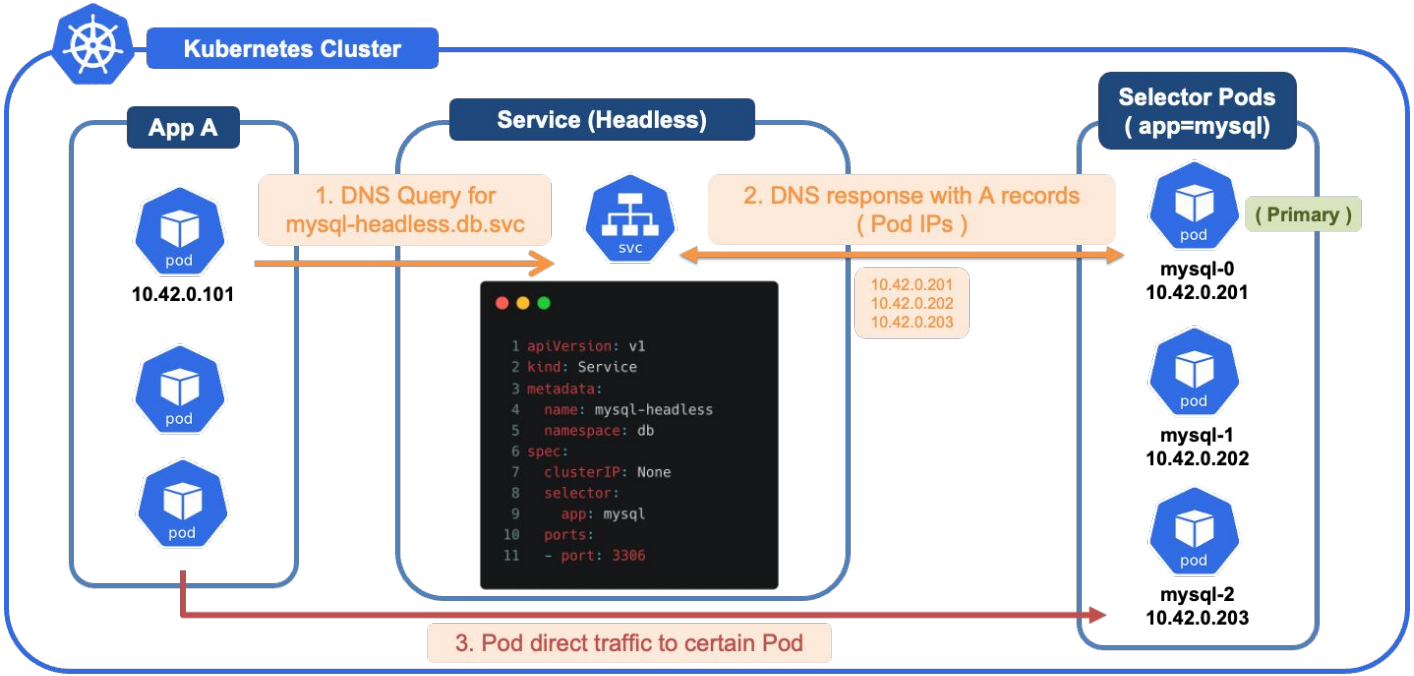
[App.] Service (svc) - ExternalName

- CNAME alias to an external DNS name (no proxying, no ClusterIP)
- Use: access external DB as a K8s Service name (migration-friendly)



[App.] Service (svc) - Headless

- No VIP; DNS returns individual Pod IPs
- Required by StatefulSet for per-Pod stable DNS

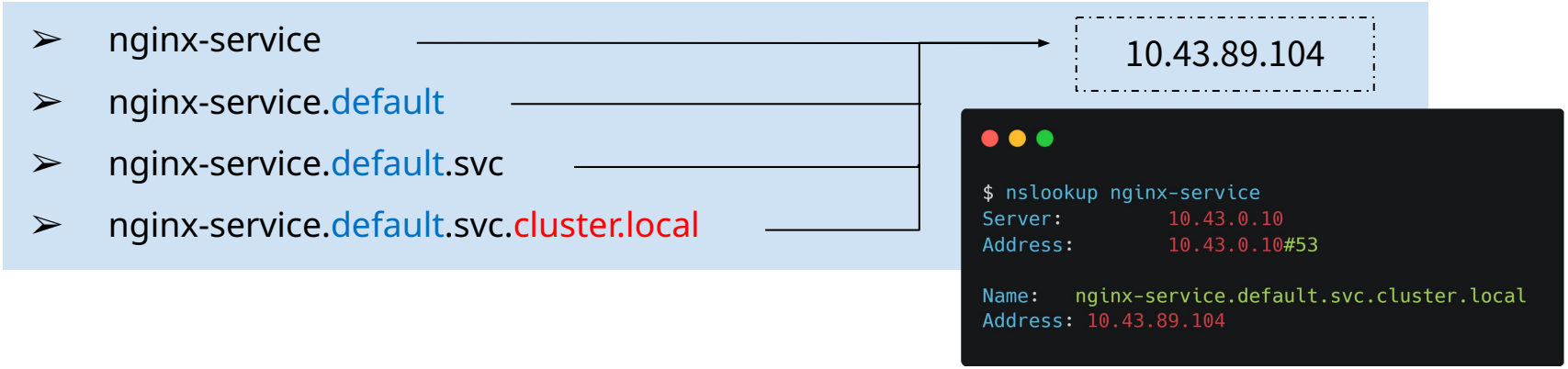


Kubernetes DNS & CoreDNS

- CoreDNS is the default cluster DNS server
  - Runs as a Deployment in kube-system namespace
  - Every Pod's /etc/resolv.conf points to CoreDNS ClusterIP
- DNS record formats
  - Service: `<serviceName>.<namespace>.svc.cluster.local` → ClusterIP (A record)
- CoreDNS config is in ConfigMap coredns in kube-system
  - `kubectl -n kube-system edit cm coredns`

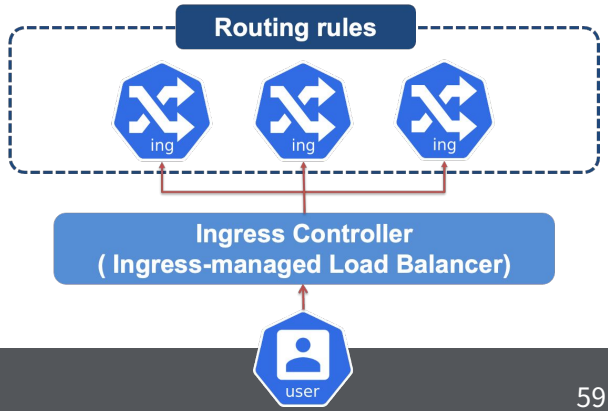
[App.] Kubernetes DNS & CoreDNS

- Short-name resolution (search domain chaining)
  - Pod in "prod" namespace can reach "api-service" without FQDN
  - resolv.conf search list adds:
    - `prod.svc.cluster.local → svc.cluster.local → cluster.local`



Ingress

- Ingress vs Service
  - Service: operate at Layer 4, one public IP per Service (expensive)
  - Ingress: operates at Layer 7, ONE entry point (Ingress contoller) routes to many Services
- Ingress refers to routing rules, manages external HTTP/HTTPS access to internal Services
  - Host-based routing: api.example.com → api-service
  - Path-based routing: /api → api-service, /web → web-service
- TLS termination: handles HTTPS, forwards plain HTTP internally
- Ingress requires an Ingress Controller to function



Ingress

- Ingress Controller – the actual implementation
  - A set of pods that watch Ingress resources and implement the routing rules
  - It acts as a reverse proxy and load balancer for HTTP/HTTPS traffic into the cluster
  - Popular Ingress Controllers include: NGINX, Traefik, HAProxy, Istio Ingress Gateway, etc.
  - Allow Multiple controllers co-exist in a cluster
    - `ingressClassName` field selects which controller handles this Ingress

[App.] Ingress - YAML Manifest

- ingressClassName
  - which controller handles this Ingress
- rules → host → paths
  - pathType: Prefix
  - backend: api-service

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-demo
spec:
  ingressClassName: nginx
  rules:
  - host: "api.example.com"
    http:
      paths:
      - pathType: Prefix
        path: "/api"
        backend:
          service:
            name: api-service
            port:
              number: 80
```

Health Checks - Probes

Health Checks

- Kubernetes Probe
  - Health Check Mechanism: A built-in diagnostic tool to monitor container status
  - Automated Monitoring: Periodically audits containers to ensure they are functioning as expected.
- kubelet performs probes to monitor container health
- Three types of probes:
  - Liveness Probe
  - Readiness Probe
  - Startup Probe

Health Checks - Three Types of Probes

- **Liveness Probe** – "Is the container still alive?"
  - Failure action: kill and restart the container
  - Use for: deadlocks, infinite loops, hung processes
  - Warning: do NOT include external dependencies
    - If your DB is down, you don't want your backend Pod restarted

Health Checks - Three Types of Probes

- **Readiness Probe** – "Is the container ready to serve traffic?"
  - Failure action: remove Pod from Service backends (no restart)
  - Traffic stops until probe passes again
  - Use for: startup warm-up, temporary overload, DB initialization
  - Critical for zero-downtime rolling updates
    - New Pod added to Service backends only after Readiness passes
- **Startup Probe** – "Has the app finished starting up?"
  - Disables Liveness + Readiness until it passes once
  - Prevents premature restarts for slow-starting apps (JVM, etc.)
    - periodSeconds x failureThreshold= max startup time allowed
- Minimum recommendation: **always** define a Readiness probe

Health Checks - Probe Mechanisms

- Four probe mechanisms
  - **httpGet**
    - HTTP request; 2xx/3xx response = success
    - Fields: path, port, httpHeaders, scheme (HTTP/HTTPS)
  - **exec**
    - run command inside container; exit code 0 = success
    - Use when there is no HTTP endpoint
  - tcpSocket
    - TCP connection established = success
    - Use for non-HTTP services: databases, gRPC, AMQP
  - grpc
    - gRPC Health Checking Protocol (K8s 1.24+)

[App.] Health Checks - Parameters

- Configuration parameters
  - initialDelaySeconds – wait before first probe (default 0)
  - periodSeconds – probe interval (default 10s)
  - timeoutSeconds – probe timeout (default 1s)
  - failureThreshold – consecutive failures before action (default 3)
  - successThreshold – consecutive successes to recover (default 1)
- Common pitfalls
  - initialDelaySeconds too short → repeated probe failures / restart loop during startup
  - timeoutSeconds too short → flapping probes under load
  - No Readiness probe → 502/503 errors during rolling update

[App.] Health Checks – YAML Manifest

- readinessProbe controls traffic
- livenessProbe controls restart

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: healthcheck-demo
spec:
  containers:
    - name: healthcheck-demo
      image: 'registry.k8s.io/busybox:1.27.2'
      args:
        - /bin/sh
        - '-c'
        - touch /tmp/healthy; sleep 30; rm -f /tmp/healthy; sleep 600
      livenessProbe:
        exec:
          command:
            - cat
            - /tmp/healthy
        initialDelaySeconds: 5
        periodSeconds: 5
      readinessProbe:
        exec:
          command:
            - cat
            - /tmp/healthy
        initialDelaySeconds: 5
        periodSeconds: 5
```

Resource Management

Including: Requests & Limits, QoS, LimitRange / ResourceQuota

Resource Requests & Limits

- Two resource types: CPU (compressible) and Memory (incompressible)
  - CPU units
    - 1 = 1 vCPU core; 1000m = 1 core; 250m = 0.25 core
  - Memory units
    - Ki / Mi / Gi (binary, base-1024) vs K / M / G (decimal)
    - Use Mi / Gi to avoid ambiguity
- **requests** – minimum guaranteed; used by the Scheduler
  - Pod is only placed on a Node that has enough free requested resources
  - Does NOT mean the Pod is capped at this amount
- **limits** – hard maximum; enforced by Linux cgroups at runtime

Resource Requests & Limits

- Behavior when limits are exceeded
  - CPU: container is THROTTLED – slowed down, NOT killed
    - CPU throttling can cause unexpected latency spikes
  - Memory: container is OOMKilled – kernel kills it immediately
    - `kubectl describe pod` or container last state shows OOMKilled
    - Container restarts if restartPolicy: Always
- Tip: set CPU limit with care; CPU throttling is often misdiagnosed

Quality of Service Classes (QoS)

- Quality of Service class determines eviction priority under memory pressure
- **Guaranteed** – requests == limits for ALL containers in Pod
  - Highest priority, last to be evicted
  - Best for latency-sensitive, stateful workloads
- **Burstable** – at least one container has requests ≠ limits
  - Middle priority, evicted after BestEffort and before Guaranteed
- **BestEffort** – no resource requests or limits set
  - Lowest priority, evicted first under pressure
  - Never set in production

LimitRange and ResourceQuota

- **LimitRange** – namespace-level defaults & bounds
  - Sets default requests/limits for Pods that omit them
  - Enforces min/max per container (rejects Pods that violate)
- **ResourceQuota** – namespace-level total cap
  - Limits total CPU / memory / PVCs / object count per namespace
  - Prevents one team from starving the entire cluster
    - hard: { requests.cpu: "10", limits.memory: "40Gi", pods: "50" }

[App.] Resource Requests & Limits - YAML Manifest

- Guaranteed QoS
  - requests == limits
- LimitRange sets
  - namespace defaults
  - applied when Pod omits resources

```yaml
containers:
  - name: app
    resources:
      requests:
        cpu: 500m
        memory: 512Mi
      limits:
        cpu: 500m
        memory: 512Mi
```

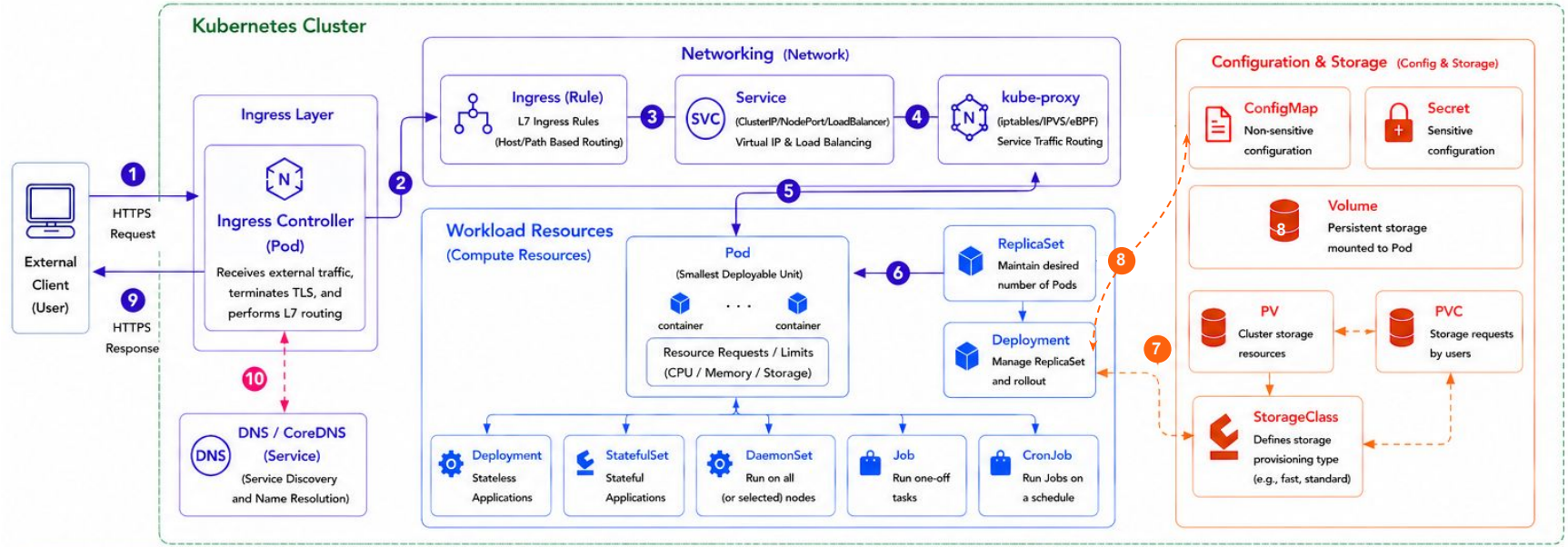
```yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: default-limits
  namespace: dev
spec:
  limits:
    - type: Container
      default:
        cpu: 200m
        memory: 256Mi
      defaultRequest:
        cpu: 100m
        memory: 128Mi
      max:
        cpu: '2'
        memory: 2Gi
```

[App.] Resource Requests & Limits - YAML Manifest

- ResourceQuota
  - hard limits total usage in namespace
  - enforced at create time, not runtime

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: team-quota
  namespace: dev
spec:
  hard:
    pods: '50'
    requests.cpu: '10'
    requests.memory: 20Gi
    limits.cpu: '20'
    limits.memory: 40Gi
    persistentvolumeclaims: '10'
```

Summary


Kubernetes Cluster
Networking (Network)
Ingress (Rule)
L7 Ingress Rules
(Host/Path Based Routing)
Service
(ClusterIP/NodePort/LoadBalancer)
Virtual IP & Load Balancing
kube-proxy
(iptables/IPVS/eBPF)
Service Traffic Routing
Ingress Layer
Ingress Controller
(Pod)
Receives external traffic, terminates TLS, and performs L7 routing
External Client (User)
HTTPS Request
HTTPS Response
Workload Resources
(Compute Resources)
Pod
(Smallest Deployable Unit)
container
container
Resource Requests / Limits
(CPU / Memory / Storage)
ReplicaSet
Maintain desired number of Pods
Deployment
Manage ReplicaSet and rollout
Deployment
Stateless Applications
StatefulSet
Stateful Applications
DaemonSet
Run on all (or selected) nodes
Job
Run one-off tasks
CronJob
Run Jobs on a schedule
DNS / CoreDNS
(Service)
(Service Discovery and Name Resolution)
Configuration & Storage (Config & Storage)
ConfigMap
Non-sensitive configuration
Secret
Sensitive configuration
Volume
Persistent storage mounted to Pod
PV
Cluster storage resources
PVC
Storage requests by users
StorageClass
Defines storage provisioning type (e.g., fast, standard)

[App.] Request workflow

1. **User sends request**
   - User initiates an HTTP/HTTPS request to the application domain/IP
2. **Request enters Ingress Controller**
   - Traffic reaches the Ingress Controller which terminates TLS and applies Ingress rules
3. **Ingress routes to Service**
   - Ingress Controller matches the rule and forwards the request to the target Service (ClusterIP)
4. **Service load balances**
   - Service selects a healthy backend Pod IP and forwards the traffic via kube-proxy
5. **Traffic reaches Pod**
   - kube-proxy (iptables/IPVS/eBPF) routes the traffic to the selected Pod through the CNI network

[App.] Request workflow

6. **Pod processes request**
   - The containers inside the Pod process the application request within resource limits
7. **Storage access (if needed)**
   - Pod reads/writes data using Volumes backed by PV/PVC (if the application needs persistent storage)
8. **Configuration access (if needed)**
   - Pod reads configuration data from ConfigMap and/or Secret
9. **Response back to user**
   - The response travels back: Pod → Service → Ingress Controller → User
10. **DNS resolution (if needed)**
    - If the user accesses via domain, CoreDNS resolves the domain name to the Ingress IP