

# Kubernetes Patterns

Reusable Elements for Designing Cloud-Native Applications (*Bilgin Ibryam and Roland Huß*)

---

## Chapter 1: Introduction

The path to cloud native, where the microservices architectural style is highlighted as a popular approach for cloud-native platforms like Kubernetes. This involves breaking down complex software into smaller, independent business capabilities. While domain-driven design and related concepts are important for designing microservices, there are also significant technical concerns related to their organisation and runtime behaviour in distributed systems.

Containers and container orchestrators like Kubernetes provide new tools and ways to handle these distributed application concerns. While Kubernetes offers many benefits, the quality of the application inside the containers is crucial; *putting "rubbish" in will result in "distributed rubbish"*.

They also mention that creating good cloud-native applications requires a mix of skills, including domain-driven design, microservices architecture understanding, and container best practices. The book itself focuses on the patterns and practices related to container orchestration, assuming the underlying application design is sound.

The chapter then delves into distributed primitives by drawing a comparison with object-oriented programming (OOP) concepts, specifically using Java as an example. The goal is to make the new Kubernetes abstractions more relatable.

The below table provides a direct comparison between local (OOP) and distributed (Kubernetes) primitives:

Concept	Local primitive	Distributed primitive
Behavior encapsulation	Class	Container image
Behavior instance	Object	Container
Unit of reuse	<i>.jar</i>	Container image
Composition	Class A contains Class B	Sidecar pattern
Inheritance	Class A extends Class B	A container's FROM parent image
Deployment unit	<i>.jar/.war/.ear</i>	Pod
Buildtime/Runtime isolation	Module, Package, Class	Namespace, Pod, container
Initialization preconditions	Constructor	Init container
Postinitialization trigger	Init-method	postStart
Predestroy trigger	Destroy-method	preStop
Cleanup procedure	<code>finalize()</code> , shutdown hook	Defer container <sup>3</sup>
Asynchronous & parallel execution	<code>ThreadPoolExecutor</code> , <code>ForkJoinPool</code>	Job
Periodic task	<code>Timer</code> , <code>ScheduledExecutorService</code>	CronJob
Background task	Daemon thread	DaemonSet
Configuration management	<code>System.getenv()</code> , <code>Properties</code>	<code>ConfigMap</code> , <code>Secret</code>

A Pod is likened to an Inversion of Control (IoC) context where multiple containers share a managed lifecycle and can communicate directly. Containers are fundamental in Kubernetes, and creating modular, reusable, single-purpose container images is key. They suggest viewing containers as:

- Units of deployment and execution

- Units of resource isolation
- Portable and immutable artifacts
- Composable building blocks

Pods are introduced as the smallest deployable units in Kubernetes, often corresponding to a single microservice (typically one container image). However, a Pod can contain multiple co-located containers that are tightly coupled and share resources.

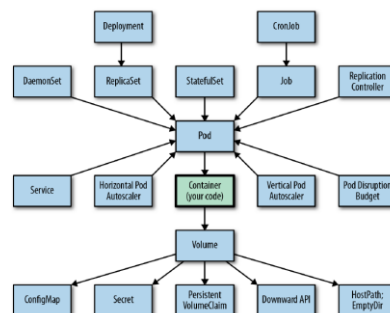
The concept of Services is then introduced as another crucial Kubernetes abstraction.

A Service permanently binds a Service name to an IP address and port number, acting as a named entry point for accessing an application, which is often a set of Pods.

Services enable Service discovery and load balancing, allowing for implementation changes and scaling without affecting consumers.

To manage and identify groups of Pods (especially when an application is split into multiple microservices), labels are used. Label selectors allow querying and managing a set of Pods as a single logical unit, providing an application identity.

Visual overview of the multitude of Kubernetes resources.



## Part I- Foundational Patterns

### Chapter 2 Predictable Demands

- The foundation for successfully deploying, managing, and ensuring applications can co-exist in a shared cloud environment relies on identifying and declaring the application's resource requirements and runtime dependencies. This is the core of the Predictable Demands pattern.
- Declaring requirements is essential for Kubernetes to find the right place for your application within the cluster.
- Problem: Kubernetes needs to know the application's needs (runtime dependencies and resource requirements) to schedule and manage it effectively. Without this information, Kubernetes treats containers as opaque boxes.
- Applications should declare their requirements, including both hard runtime dependencies and resource needs (like CPU and memory).

- Runtime Dependencies:
  - Applications often depend on external services or configuration. Kubernetes allows declaring dependencies on resources within the cluster.

For example,

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
    - image: k8spatterns/random-generator:1.0
      name: random-generator
      env:
        - name: PATTERN
          valueFrom:
            configMapKeyRef:
              name: random-generator-config
              key: pattern
```

NOTE: Pod that depends on a ConfigMap named random-generator-config. The valueFrom field in the environment variable declaration indicates this dependency. The container will only start if this ConfigMap is present in the same namespace.

- Resource Requirements:
  - It's important to distinguish between **compressible resources** (like CPU) and **incompressible resources** (like memory). If a container uses too much CPU, it gets throttled; if it uses too much memory, it gets killed
  - Applications should specify the minimum amount of resources needed (requests) and the maximum amount it can use (limits) for CPU and memory. This is like soft and hard limits.

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
    - image: k8spatterns/random-generator:1.0
      name: random-generator
      resources:
        requests:
          cpu: 100m
          memory: 100Mi
        limits:
          cpu: 200m
          memory: 200Mi
```

- Requests vs. Limits:
  - *The requests amount is used by the Kubernetes scheduler to decide which nodes have enough capacity to accommodate the Pod.* The scheduler sums up the requested resources of all containers in a Pod when making placement decisions.

- *The limits ensure that a container cannot consume more than the specified amount of resources.* If it tries to, Kubernetes will take action (e.g., throttling CPU, killing the container for exceeding memory).

**Limit Range:** This Kubernetes feature allows administrators to set resource usage limits for each type of resource within a namespace. It can enforce minimum and maximum permitted amounts, default values for requests and limits, and control the ratio between requests and limits (overcommit level).

- **Capacity Planning Scenario:**
  - Capacity planning for a Kubernetes cluster is complex because containers can have different resource profiles and varying numbers of instances across different environments.
  - On non-production clusters, you might have mainly Best-Effort and Burstable containers for better hardware utilisation, where containers being killed due to resource starvation is not critical.
  - On production clusters, you'd likely have more Guaranteed and some Burstable containers for stability and predictability. A container being killed here usually indicates a need to increase cluster capacity.
- How to calculate the total resource demands for all services in an environment?

Pod	CPU request	CPU limit	Memory request	Memory limit	Instances
A	500m	500m	500Mi	500Mi	4
B	250m	500m	250Mi	1000Mi	2
C	500m	1000m	1000Mi	2000Mi	2
D	500m	500m	500Mi	500Mi	1
<b>Total</b>	<b>4000m</b>	<b>5500m</b>	<b>5000Mi</b>	<b>8500Mi</b>	<b>9</b>

*Importance of Resource Profiles: Resource profiles (requests and limits) are how an application communicates its needs to Kubernetes, assisting in scheduling and management. If these are not provided, Kubernetes can only treat containers as opaque boxes and may drop them when the cluster is full.*

- Providing resource declarations is more or less mandatory for every application running on Kubernetes.
- Resource profiles help Kubernetes make informed decisions about placing Pods on suitable nodes and managing resource contention. If a container consume too many compressible resources (CPU), they are throttled, but **if a container tries to use too many incompressible resources then then they are killed.**
  - **Pod Priority Class:** Pod Priority is a user-specified numerical priority, with high-priority has a priority of 1,000. When multiple Pods are waiting to be placed, the scheduler sorts the queue of pending Pods by highest priority first. If there are no nodes with enough capacity to place a Pod, the scheduler can preempt (remove) lower-priority Pods from nodes to free up resources and place Pods with higher priority. **The Kubelet first considers QoS and then PriorityClass of Pods before eviction.**

- The pod that does not have request or limit set for a container, they are consider as lowest priority and most likely killed first. **(These types of pods are called as best-effort pods)**
- The pod that has limit set higher than the request, they are called as **Burstable** pods, such a Pod has minimal resource guarantees, but is also willing to consume more resources up to its limit when available. When the node is under incompressible resource pressure, these Pods are likely to be killed if no Best-Effort Pods remain.

```
##The definition of the PriorityClass value
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority          #Name
  value: 1000                  #Value
globalDefault: false
description: This is a very high priority Pod class
---
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
  labels:
    env: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    priorityClassName: high-priority #Name Referred here
```

## Chapter 3 Declarative Deployment

- The heart of the Declarative Deployment pattern is the Kubernetes Deployment resource. This resource automates the upgrade and rollback processes for a group of containers.
- Problem: Manually updating applications can be complex, error-prone, difficult to document, and hard to track the desired state.
- Solution: Kubernetes automates application updates through the Deployment concept, using different strategies. This saves effort by automating multiple deployments per release cycle.
- A Deployment relies on well-behaved containers that honour lifecycle events and provide health-check endpoints.
- Declarative updates using the Deployment resource are preferred over imperative updates.
- **Rolling Deployment:**
  - This is the default update strategy in Kubernetes.
  - It updates applications gradually by replacing old instances with new ones while ensuring zero downtime.
  - Example below shows a deployment configuration for a rolling update, specifying
    - number of replicas

- `strategy.type: RollingUpdate`
  - `rollingUpdate.maxSurge` (defines the number of additional Pods that can run temporarily during an update)
  - `rollingUpdate.maxUnavailable` (defines the number of Pods that can be unavailable during the update)
  - Readiness probes are crucial for zero-downtime rolling deployments to ensure new Pods are ready before old ones are terminated.
  - Updates can be triggered by replacing the Deployment, patching it, editing it, or using `kubectl set image`.
- Benefits include server-side management, declarative definition of the desired state, executable and tested deployment definitions, and recorded/versioned update processes with rollback options.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: random-generator
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
  selector:
    matchLabels:
      app: random-generator
  template:
    metadata:
      labels:
        app: random-generator
    spec:
      containers:
        - image: k8spatterns/random-generator:1.0
          name: random-generator
      readinessProbe:
        exec:
          command: [ "stat", "/random-generator-ready" ]

```

- **Fixed Deployment (using Recreate strategy):**
  - This strategy sets `strategy.type` to `Recreate`.
  - It first kills all existing containers of the old version and then starts all new containers simultaneously.
  - This results in downtime but ensures that only one version of the application is running at any time.
  - This is useful when updates introduce backward-incompatible API changes.
- **Blue-Green Release:**
  - This strategy minimises downtime and risk by running two identical production environments (blue and green), with only one live at a time.
  - Kubernetes Deployments, along with other primitives, can be used to implement this.

- Only one version serves requests, simplifying complexity for service consumers.
- The downside is the requirement for double the application capacity and potential complications with long-running processes and database state during transitions.
- **Canary Release:**
  - This technique deploys a new version to a small subset of users before a full rollout.
  - In Kubernetes, this can be achieved by creating a new ReplicaSet (preferably via a Deployment) with a small replica count for the new version.
  - The Service directs some traffic to these canary instances.
  - After verifying the new version, the new ReplicaSet is scaled up, and the old one is scaled down.
  - This allows for a controlled and user-tested incremental rollout.

#### Discussion:

- The Deployment primitive automates manual application updates into a repeatable and automated declarative activity.
- Out-of-the-box strategies (**rolling** and **recreate**) manage container replacement, while release strategies (**blue-green** and **canary**) control new version availability to consumers.
- **Blue-green** and **canary** releases require human decision for the transition trigger and are not fully automated.
- The discussed techniques primarily cover Pod updates and don't inherently handle updates or rollbacks of dependencies like ConfigMaps, Secrets, or other services.
- A proposal exists for deployment hooks (pre and post) to allow custom commands during deployment, enabling more automated strategies. For now, scripting at a higher level can manage updates of services and their dependencies.
- It is crucial for Kubernetes to know when application Pods are running to perform deployment steps correctly. Health Probes (Chapter 4) allow applications to communicate their health state to Kubernetes.

NOTE: A proposal for Kubernetes to allow hooks in the deployment process. Pre and Post hooks would allow the execution of custom commands before and after Kubernetes has executed a deployment strategy.

## Chapter 4 Health Probe

- The Health Probe pattern is about how an application communicates its health state to Kubernetes. This observability is crucial for Kubernetes to automate management tasks.
- Problem: Simply checking the container process status is often not enough to determine if an application is healthy. An application process can be running but the application itself might be hung, deadlocked, or unable to serve requests. Kubernetes needs a more reliable way to assess application health beyond just the process state.
- Solution: Applications should implement specific APIs or expose certain indicators that Kubernetes can probe to understand if the application is **"up" (liveness)** and **"ready" to serve traffic (readiness)**.

- Process Health Checks:
  - This is the most basic check where the Kubelet constantly monitors the container processes. If a process isn't running, the container is restarted.
  - If your application can detect its own failures and shut down, this might be sufficient. However, this is often not the case.
  - **Liveness Probes:**
    - These probes determine if the application is running correctly. If a liveness probe fails, **Kubernetes will restart the container**. Kubernetes supports **three types** of liveness probes:
      - **HTTP probe:** Sends an HTTP GET request to a specified path and port on the container's IP address. A successful response is within the **200-399 HTTP status code range**.
      - **TCP Socket probe:** Attempts to establish a TCP connection to a specified port on the container's IP address. A successful connection indicates health.
      - **Exec probe:** Executes a specified command within the container's namespace. A successful exit code (0) indicates health.

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-liveness-check
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    env:
    - name: DELAY_STARTUP
      value: "20"
    ports:
    - containerPort: 8080
    protocol: TCP
    livenessProbe:
      httpGet:
        path: /actuator/health
        port: 8080
      initialDelaySeconds: 30
```

NOTE: The choice of probe type depends on the application's nature. It is the application's responsibility to decide what constitutes a healthy state. However, a failing liveness probe leading to constant restarts without addressing the underlying issue is not beneficial.

- **Readiness Probes:**
  - These probes determine if the application is ready to handle incoming traffic. If a readiness probe fails, **the container is removed from the service endpoints** and will not receive new traffic. Unlike liveness probes, a failed readiness probe does not trigger a container restart.
  - Readiness probes are useful during application startup (allowing warm-up time before receiving requests) and also later if a container becomes overloaded and needs to shield itself from additional load.

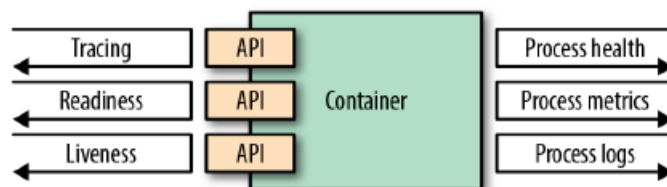


- The methods for readiness checks are the same as for liveness checks (HTTP, TCP, Exec).

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-readiness-check
spec:
  containers:
    - image: k8spatterns/random-generator:1.0
      name: random-generator
      readinessProbe:
        exec:
          command: ["stat", "/var/run/random-generator-ready"]
```

#### Discussion:

- Health checks are fundamental for automating activities like deployment, self-healing, and scaling.
- Logging is another important way for applications to provide visibility into their health by recording significant events to system out and system error for central collection and analysis. While not used for automated actions, logs are useful for alerts and postmortem analysis.



## Chapter 5 Managed Lifecycle

- The Managed Lifecycle pattern describes how containerised applications should listen to and react to lifecycle events emitted by the managing platform (Kubernetes) to be good cloud-native citizens.
- Problem: Containerised applications managed by platforms like Kubernetes have no direct control over when they are started or stopped. The platform can decide to start or stop applications at any time based on policies and external factors. Applications need to be aware of these events to perform necessary actions like warming up or shutting down gracefully.
- Solution: Applications can listen to and react to specific lifecycle events emitted by Kubernetes. This allows for more fine-grained interactions and lifecycle management capabilities beyond simply starting and stopping processes.
- **Lifecycle Events and Hooks:** Kubernetes provides mechanisms for containers to execute specific actions during their lifecycle. These are primarily lifecycle hooks:
  - postStart Hook: This hook is executed immediately after a container has been created. It runs before the container's main process starts.
  - Use cases include initialising configurations, downloading data, or any setup tasks needed before the application starts serving traffic.

```
apiVersion: v1
kind: Pod
metadata:
  name: post-start-hook
spec:
```

```

containers:
- image: k8spatterns/random-generator:1.0
  name: random-generator
  lifecycle:
    postStart:
      exec:
        command:
          - sh
          - -c
          - sleep 30 && echo "Wake up!" > /tmp/postStart_done

```

- Post/Pre Start

```

apiVersion: v1
kind: Pod
metadata:
  name: post-start-hook
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    lifecycle:
      postStart:
        exec:
          command:
            - sh
            - -c
            - sleep 30 && echo "Wake up!" > /tmp/postStart_done

```

- postStart hooks are executed asynchronously with respect to the container's startup. If the hook fails to complete, the container will eventually be killed.
- preStop Hook: This hook is executed immediately before a container is terminated. Kubernetes sends a SIGTERM signal to the container, and if a preStop hook is defined, it is executed before the container is forcibly killed with SIGKILL after a grace period.
- Use cases include gracefully shutting down the application, finishing ongoing requests, saving state, or notifying other services about the impending shutdown.
- preStop hooks are blocking, meaning Kubernetes will wait for the hook to complete before proceeding with the termination (up to the pod's terminationGracePeriodSeconds).

NOTE: The same handler types (exec or HTTP) and guarantees as postStart are offered. It is important to note that even if the preStop hook takes too long or fails, the container will still be deleted and the process killed. It is a convenience for graceful shutdown, not a guarantee against forceful termination.

Signals: Besides lifecycle hooks, containers also receive standard POSIX signals:

- **SIGTERM Signal:** This is the graceful termination signal sent by Kubernetes to the main process in the container when it needs to be stopped. Applications should handle this signal to shut down cleanly.
- **SIGKILL Signal:** If the application does not terminate gracefully within a defined grace period after receiving SIGTERM, Kubernetes will send SIGKILL, which forcibly terminates the process.

#### Other Lifecycle Controls: Init Containers:

- Init Containers are specialised containers that run before the application containers in a Pod start.
- They **run sequentially** and must complete successfully before the next init container or the application containers are started.
- Use cases include setting up the environment, installing dependencies, or performing initialisation tasks that the main application containers depend on.

Aspect	Lifecycle hooks	Init Containers
Activates on	Container lifecycle phases	Pod lifecycle phases
Startup phase action	A <code>postStart</code> command	A list of <code>initContainers</code> to execute
Shutdown phase action	A <code>preStop</code> command	No equivalent feature exists yet
Timing guarantees	A <code>postStart</code> command is executed at the same time as the container's <code>ENTRYPOINT</code>	All init containers must be completed successfully before any application container can start
Use cases	Perform noncritical startup/shutdown cleanups specific to a container	Perform workflow-like sequential operations using containers; reuse containers for task executions

- Lifecycle hooks activate on container lifecycle phases, while init containers activate on Pod lifecycle phases.
- `postStart` is a startup action at the container level, while init containers are a list of containers executed before any application container.
- `preStop` is a shutdown action at the container level; there's no direct equivalent for shutdown at the Pod init container level yet.
- `postStart` runs concurrently with the container's `entrypoint`, while all init containers must complete before application containers start.
- Lifecycle hooks are for non-critical startup/shutdown cleanups specific to a container. Init containers are for workflow-like sequential operations and reusing containers for task executions.
- Reacting to lifecycle events allows applications to benefit fully from the capabilities of cloud-native platforms like Kubernetes, ensuring graceful startup and shutdown.
- It shifts the mindset from manual lifecycle management to platform-automated management.
- Honoring these contracts ensures applications can run and scale reliably.

## Chapter 6 Automated Placement

- Automated Placement, focuses on how Kubernetes automatically assigns Pods to suitable nodes within a cluster using its scheduler.

- **Problem:** In a microservices-based system with many isolated processes (Containers and Pods), individually assigning and placing them onto suitable nodes becomes unmanageable.
- **Solution:** Kubernetes uses a scheduler to automatically assign Pods to nodes that meet their requirements and respect scheduling policies. This is a highly configurable and evolving area.
- **Scheduler's Role:** The scheduler is a fundamental and time-saving tool. It can run in isolation or *not be used at all*, but it plays a central role in the Kubernetes platform.
- Influencing Factors on Placement:
  - **Container Resource Demands:** Containers should have their runtime dependencies and resource needs (requests and limits) defined, Predictable Demands. This is crucial for the scheduler to make sensible placement decisions and prevent interference between Pods.
- Scheduling Process:
  - When a Pod is created without a node assignment, the scheduler selects it along with all available nodes and a set of **filtering** and **priority** policies.
  - **Filtering:** The scheduler applies filtering policies to exclude nodes that don't meet the Pod's criteria.
  - **Prioritisation:** The remaining nodes are then ranked based on weight using priority policies.
  - **Assignment:** Finally, the Pod is **assigned to the highest-ranked node**.

### Scheduler Policies

Scheduler behaviour can be influenced by policies defining predicates (filtering rules) and priorities (ranking rules). An example policy includes predicates & priorities

```
{
  "kind" : "Policy",
  "apiVersion" : "v1",
  "predicates" : [
    {"name" : "PodFitsHostPorts"},
    {"name" : "PodFitsResources"},
    {"name" : "NoDiskConflict"},
    {"name" : "NoVolumeZoneConflict"},
    {"name" : "MatchNodeSelector"},
    {"name" : "HostName"}
  ],
  "priorities" : [
    {"name" : "LeastRequestedPriority", "weight" : 2},
    {"name" : "BalancedResourceAllocation", "weight" : 1},
    {"name" : "ServiceSpreadingPriority", "weight" : 2},
    {"name" : "EqualPriority", "weight" : 1}
  ]
}
```

- **Controlling Pod Placement:** While it is generally best to let the scheduler handle placement, there are ways to influence its decisions. **[Controlling Pod Placement, cannot to changed/alterd]**
- **Node Selector:** The **.spec.nodeSelector** in a Pod definition allows specifying key-value pairs that must exist as labels on a node for it to be eligible to run the Pod.

Forcing a Pod to run on a node with SSD storage by using a nodeSelector like disktype: ssd

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
    - image: k8spatterns/random-generator:1.0
      name: random-generator
      nodeSelector:
        disktype: ssd #1
#1 Set of node labels a node must match to be considered to
be the node of this Pod
```

- **Node Affinity:** A more flexible way to configure scheduling based on node labels, allowing for both required (must be met) and preferred (weighted preference) rules. It also offers more expressive operators like **In**, **NotIn**, **Exists**.

A **requiredDuringSchedulingIgnoredDuringExecution** rule requiring a node to have more than three cores (based on a node label) and a **preferredDuringSchedulingIgnoredDuringExecution** rule with weights for nodes matching certain label criteria. The required rule is a hard constraint, while the preferred rule adds a score to matching nodes.

**Pod Affinity and Antiaffinity:** These allow defining rules based on the labels of already running Pods on nodes or other topology domains (like racks or cloud zones). This enables co-location or spreading of related Pods.

The below Pod definition with podAffinity requiring Pods with the label confidential: high to be in the same security zone and podAntiAffinity (as a preferred rule with a weight) to avoid running on the same hostname as Pods with the label confidential: none.

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution: #1
      - labelSelector: #2
        matchLabels:
          confidential: high
        topologyKey: security-zone #3
    podAntiAffinity: #4
      preferredDuringSchedulingIgnoredDuringExecution: #5
      - weight: 100
        podAffinityTerm:
          labelSelector:
            matchLabels:
              confidential: none
          topologyKey: kubernetes.io/hostname
  containers:
    - image: k8spatterns/random-generator:1.0
      name: random-generator
```

```
# 1 Required rules for the Pod placement concerning other Pods running on
the target node.

#2 Label selector to find the Pods to be colocated with.

#3 The nodes on which Pods with labels confidential=high are running are
supposed to carry a label security-zone. The Pod defined here is scheduled
to a node with the same label and value.

#4 Antiaffinity rules to find nodes where a Pod would not be placed.

#5 Rule describing that the Pod should not (but could) be placed on any
node where a Pod with the label confidential=none is running.
```

- **Taints and Tolerations:** Taints are applied to nodes and prevent Pods from being scheduled on them unless the Pod has a matching toleration. This is useful for dedicating nodes for specific workloads.
- **Descheduler:** This tool identifies and evicts Pods from nodes to improve resource utilisation and cluster balance. Strategies include LowNodeUtilization, which evicts Pods from overutilised nodes to be placed on underutilised ones. Eviction policies respect Pod Quality of Service (QoS) levels, preferring to evict Best-Effort, then Burstable, and lastly Guaranteed Pods.

#### Discussion:

The general recommendation is to intervene in the scheduling process as little as possible. By properly sizing and declaring container resource needs and using labels effectively, the scheduler usually makes appropriate placement decisions. However, when necessary, the described mechanisms offer increasing levels of control over Pod scheduling. More advanced options include using multiple custom schedulers with different policies for subsets of Pods.

Key Takeaway: Properly define container resource profiles, label Pods and nodes, and only minimally intervene with the Kubernetes scheduler to achieve the desired deployment topology.

## Part II- Behavioral Patterns

Behavioral pattern focuses on the communication mechanisms and interaction between the Pods and the managing platform. Depending on the type of managing controller, a Pod may run until completion.

### Chapter 7 Batch Job

- **Problem:** Managing and running finite, isolated units of work reliably until completion on a distributed environment becomes necessary. Unlike long-running processes managed by Deployments or ReplicaSets, some tasks need to run once and then stop.

*Bare Pod: It is possible to create a Pod manually to run containers. However, when the node such a Pod is running on fails, the Pod is not restarted.*

*ReplicaSet : This controller is used for creating and managing the lifecycle of Pods expected to run continuously.*

*DaemonSet : A controller for running a single Pod on every node. Typically used for managing platform capabilities such as monitoring, log aggregation, storage containers, and others.*

The Job abstraction is a fundamental primitive that other primitives like CronJob are built upon. Jobs help make isolated work units reliable and scalable. Using Jobs for short-lived tasks saves resources compared to long-running abstractions. Implementing complex batch processing often requires combining the Job primitive with a batch application framework (e.g., Spring Batch, JBeret). Jobs are scheduled on nodes with the required capacity and respecting Pod placement policies.

- **Solution:** Kubernetes provides the Job resource for managing these **short-lived, finite workloads**. A Job creates one or more Pods and ensures they run successfully to completion. Once the specified number of Pods terminate successfully, the Job is considered complete, and no new Pods are started.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: random-generator
spec:
  completions: 5 # Job should run five Pods to completion, which all must succeed.
  parallelism: 2 # indicates that two Pods can run in parallel.
  template:
    metadata:
      name: random-generator
    spec:
      restartPolicy: OnFailure # Specifying the restartPolicy
      containers:
        - image: k8spatterns/random-generator:1.0
          name: random-generator
          command:
            - "java"
              "-cp"
              "/"
              "RandomRunner"
              "/numbers.txt"
              "10000"
```

- **Single Pod Job:** When both `.spec.completions` and `.spec.parallelism` are left out or set to their default value of one. This starts only one Pod, and the Job completes when that single Pod terminates successfully (exit code 0).

◦

- Fixed completion count Jobs: When `.spec.completions` is set to a number greater than one. The Job is complete after that many Pods have finished successfully. `.spec.parallelism` can be optionally set (default is one). Example 7-1 demonstrates this type, useful when the number of work items is known in advance.
- Work queue Jobs: When `.spec.completions` is left out and `.spec.parallelism` is set to an integer greater than one. The Job is considered complete when at least one Pod has terminated successfully, and all other Pods have also terminated. This requires Pods to coordinate to determine what each is working on. Suitable for scenarios with a fixed but unknown number of work items in a queue. If there's an unlimited stream of work, ReplicaSets are a better choice.

Scenarios for Mapping Individually Processable Work Items to Jobs/Pods:

- One Job per work item: Each work item triggers the creation of a separate Kubernetes Job. This has the overhead of managing many Job resources but is useful when each work item is a complex task that needs independent recording, tracking, or scaling.
- One Job for all work items: A single Job is created to process a large number of work items. The management of these work items happens within the application itself using a batch framework. Suitable when work items don't need independent tracking by the platform.

## Chapter 8 Periodic Job

Problem: While modern systems favour real-time and event-driven interactions, periodic job scheduling is still crucial for automating system maintenance, administrative tasks, and specific business application needs.

Traditional methods like specialized scheduling software or Cron on a single server can be expensive for simple cases and present a single point of failure.

Implementing scheduling within applications (e.g., using libraries like Quartz in Java) can lead to complexity in ensuring resilience and high availability of the scheduler itself, often requiring leader election mechanisms and high resource consumption.

Solution:

- Kubernetes provides the CronJob primitive, which builds upon the Job resource (described in Chapter 7) to manage the temporal aspects of a job.
- A CronJob allows you to execute a Job periodically at a specified point in time, like a line in a Unix crontab.
- This allows developers to focus on the business logic to be performed rather than the scheduling mechanism.



```

apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: random-generator
spec:
  schedule: "*/3 * * * *" # Every three minutes
  startingDeadlineSeconds: 3 #Defines a deadline (in
whole seconds) for starting the Job, if that Job misses
its scheduled time for any reason.
  concurrencyPolicy: Allow | Forbid | Replace
  successfulJobsHistoryLimit : 3 #default value
  failedJobsHistoryLimit: 1 #default value
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - image: k8spatterns/random-generator:1.0
              name: random-generator
              command: ["java", "-cp", "/", "RandomRunner",
"/numbers.txt", "10000" ]
              restartPolicy: OnFailure

```

- **Example CronJob Resource:**
  - **schedule:** "\*/3 \* \* \* \*": This is the Cron specification that defines the job to run every three minutes.
  - **jobTemplate:** This defines the specification of the Job that will be created and executed according to the schedule. It uses the same .spec.template.spec as a regular Job, including the container image, command, and restartPolicy. The restartPolicy is set to OnFailure.

**Concurrency policy :** The .spec.concurrencyPolicy field is also optional. It specifies how to treat concurrent executions of a Job that is created by this CronJob. The spec may specify only one of the following concurrency policies:

- **Allow** (default): The CronJob allows concurrently running Jobs
- **Forbid:** The CronJob does not allow concurrent runs; if it is time for a new Job run and the previous Job run hasn't finished yet, the CronJob skips the new Job run. Also note that when the previous Job run finishes, .spec.startingDeadlineSeconds is still taken into account and may result in a new Job run.
- **Replace:** If it is time for a new Job run and the previous Job run hasn't finished yet, the CronJob replaces the currently running Job run with a new Job run

Note that concurrency policy only applies to the Jobs created by the same CronJob. If there are multiple CronJobs, their respective Jobs are always allowed to run concurrently.

**Schedule suspension:** You can suspend execution of Jobs for a CronJob, by setting the optional `.spec.suspend` field to true. The field defaults to false.

This setting does *not* affect Jobs that the CronJob has already started.

If you do set that field to true, all subsequent executions are suspended (they remain scheduled, but the CronJob controller does not start the Jobs to run the tasks) until you unsuspend the CronJob.

#### Scenarios for Periodic Jobs:

- System maintenance tasks: Automating routine maintenance operations.
- Administrative tasks: Scheduling regular administrative functions.
- Business-to-business integration: File transfers at specific intervals.
- Application integration: Database polling on a schedule.
- Communication: Sending newsletter emails periodically.
- Data management: Cleaning up and archiving old files.

#### Discussion and Benefits:

- CronJob is a simple primitive that adds clustered, Cron-like behaviour to Kubernetes.
- Combined with other Kubernetes features like Pods, resource isolation, and automated placement (Chapter 6), it becomes a powerful job scheduling system.
- Developers can focus on the business logic of the task.
- Scheduling is managed by the platform, providing benefits like high availability, resiliency, capacity management, and policy-driven Pod placement.
- When implementing a container for a CronJob, applications must consider potential issues like duplicate runs, no runs, parallel runs, or cancellations.
- CronJob demonstrates how Kubernetes capabilities build on each other to support even non-cloud-native use cases.

## Chapter 9 Daemon Service

- The Daemon Service pattern is primarily used by administrators to place and run prioritised, infrastructure-focused Pods on targeted nodes. It helps manage node-specific Pods that enhance the Kubernetes platform capabilities.
- The problem it addresses is the need for long-running, self-recovering computer programs that run as background processes. These are similar to daemons found in operating systems (like `httpd` or `sshd`) or daemon threads in applications (like Java Garbage Collector). These programs typically don't interact with standard input/output devices and start automatically with the system.

- Kubernetes provides the **DaemonSet resource** to implement this pattern. A ***DaemonSet is a controller designed to run a single Pod on every node, or on specific nodes you choose.***
- Unlike controllers driven by consumer load (like ReplicaSet), a ***DaemonSet's main purpose is to ensure a single Pod is always running on its target nodes, regardless of external demand.***
- DaemonSets are commonly used for infrastructure-related processes, such as:
  - Log collectors.
  - Metric exporters.
  - Storage containers.
  - Even core Kubernetes components like *kube-proxy*.
- Applications where the Pod pushes data externally rather than being reached by consumers.
- Example of a sample DaemonSet definition, which generated random number and write to /host\_dev mount location.

```

apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: random-refresher
spec:
  selector:
    matchLabels:
      app: random-refresher
  template:
    metadata:
      labels:
        app: random-refresher
    spec:
      nodeSelector:
        feature: hw-rng
      containers:
        - image: k8spatterns/random-generator:1.0
          name: random-generator
          command:
            - sh
            - -c
            - >-
              "while true; do
                java -cp / RandomRunner
                /host_dev/random 100000;
                sleep 30; done"
          volumeMounts:
            - mountPath: /host_dev
              name: devices
      volumes:
        - name: devices

```

```
hostPath:
  path: /dev
```

- Difference Between DeamonSet and ReplicaSet

Feature	DaemonSet	ReplicaSet
<b>Purpose</b>	Runs pods on every node (or selected nodes)	Maintains a specified number of identical pods
<b>Pod distribution</b>	One pod per node	Pods distributed anywhere in the cluster
<b>Scaling behaviour</b>	Automatically scales with node count	Manually scaled by changing replica count
<b>Node affinity</b>	Built-in node affinity	No inherent node affinity
<b>Pod creation timing</b>	Creates pods when nodes join the cluster	Creates pods immediately based on replica count
<b>Pod removal</b>	Removes pods when nodes leave the cluster	Removes pods when replica count is decreased
<b>Use cases</b>	Monitoring agents, log collectors, networking plugins	Stateless applications, web servers, microservices
<b>Node selection</b>	Can target specific nodes with nodeSelector/affinity	No default node targeting
<b>Created by</b>	Directly or through Helm charts	Directly, through Deployments, or Helm charts
<b>Rolling updates</b>	Supports rolling updates	Usually managed by parent Deployment
<b>Pod uniqueness</b>	One pod per node, each can have node-specific data	All pods are identical replicas
<b>Priority</b>	Pods managed by a DaemonSet are supposed to run only on targeted nodes, and as a result, are treated with higher priority and differently by many controllers.	Pods are created post scheduler is created.
<b>Use Of Scheduler</b>	Since the scheduler is not used, the unschedulable field of a node is not respected by the DaemonSet controller.  The descheduler will avoid evicting such Pods, the cluster autoscaler will manage them separately	ReplicaSet are managed by scheduler.

- Typically, a DaemonSet creates a single Pod on every node or subset of nodes. There are several ways to reach the pod created by the DaemonSet.
  - Creating a service with same Pod Selector as that of the DaemonSet, use the Service to reach a daemon Pod load-balanced to a random node.
  - Creating a Headless Service with same Pod selector as that of DaemonSet that can be used to retrieve multiple A records from DNS containing all Pod IPs and ports.
  - Pods in the DaemonSet can specify a hostPort and become reachable via the node IP addresses and the specified port. Since the combination of hostIp and hostPort and protocol must be unique, the number of places where a Pod can be scheduled is limited.
- **Static Pods:** Static Pods are a unique type of pod in Kubernetes that are managed directly by the kubelet daemon on a specific node, without the involvement of the Kubernetes API server or controller.

#### **Key Characteristics of Static Pods**

- Direct kubelet management: Created and managed by the kubelet on a specific node, not via the Kubernetes API server
- Node-bound: Tied to a specific node and cannot be rescheduled elsewhere
- Configuration: Defined by pod manifest files placed in a specific directory on the node (usually /etc/kubernetes/manifests/)
- Mirror pods: The kubelet creates "mirror pods" in the API server to make static pods visible, but these are read-only
- Naming convention: Static pods have the node hostname appended to their name (e.g., kube-scheduler-master-node)
- Deletion: Can only be deleted by removing their manifest file from the watched directory

#### **Common Use Cases**

- Kubernetes control plane components: Often deployed as static pods, including:
  - kube-apiserver
  - kube-controller-manager
  - kube-scheduler
  - etcd
- Node-level critical components: Components that must run before the Kubernetes cluster is fully functional

#### **Limitations**

- Cannot use ConfigMaps or Secrets (as they require API server)
- Cannot be controlled by Deployments, DaemonSets, or other controllers
- Cannot use Kubernetes service accounts
- No rescheduling capabilities
- Limited health checks and restart policies

## How to create a Static POD

1. Create a pod manifest file
2. Place it in the kubelet's configured static pod directory
3. The kubelet will automatically detect and create the pod

Sample pod manifest file.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-static-pod
  labels:
    app: nginx-static
spec:
  containers:
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80
    resources:
      limits:
        memory: "128Mi"
        cpu: "200m"
      requests:
        memory: "64Mi"
        cpu: "100m"
    livenessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 15
      timeoutSeconds: 2
      periodSeconds: 10
      failureThreshold: 3
    readinessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 5
  restartPolicy: Always
  terminationGracePeriodSeconds: 30
```

## Chapter 10 Singleton Service

- The Singleton Service pattern is used to ensure that only one instance of an application is active at a time, while still being highly available.
- The problem this pattern addresses is the need for applications or components where running multiple instances simultaneously would cause issues, such as

duplicate task execution, redundant polling, or disordered message consumption from a broker.

- While Kubernetes makes it easy to scale applications by running multiple Pod instances (replicas) to increase throughput and availability, some scenarios specifically require only one active instance.
- Kubernetes provides mechanisms to implement this pattern, which can be handled from within the application or delegated fully to Kubernetes.

Solutions and Mechanisms:

- **Out-of-Application Locking (Delegating to Kubernetes):**

- The primary way to achieve a singleton in Kubernetes is by starting a Pod and backing it with a controller that ensures a single replica.
- A ReplicaSet configured with replicas: 1 can be used. This turns the singleton Pod into a highly available singleton by ensuring at least one instance is always running and healing the Pod in case of failures.
- Important Caveat: *ReplicaSets favour availability over consistency*. While configured for one replica, a ReplicaSet guarantees at least one instance, and occasionally more instances can run simultaneously, especially during failures or transitions.
- ***For strict singleton requirements that favour consistency over availability and provide "at most" semantics, the StatefulSet resource is recommended.*** StatefulSets are designed for stateful applications and offer stronger guarantees, although they come with increased complexity.

To enable access to a singleton Pod (especially one managed by a StatefulSet) that needs to accept incoming connections, a Kubernetes Service is used.

For a Pod managed by a StatefulSet, a headless Service (clusterIP: None) is often preferred. This type of Service doesn't get a virtual IP and doesn't perform load balancing but creates DNS A records pointing directly to the Pod's IP address(es), allowing clients to access the single Pod instance directly and predictably.

Example: For non-strict singletons, a ReplicaSet with one replica and a regular Service suffices. For strict singletons and better performance, a StatefulSet and a headless Service are preferred.

- **In-Application Locking (Application-Managed):**

- In a distributed environment, the application itself can use a distributed lock mechanism (like Apache ZooKeeper, HashiCorp's Consul, Redis, or Etcd) to ensure only one instance is active.
- Instances attempt to acquire the lock; only the instance that succeeds becomes active, while others remain passive and wait. This provides an active-passive failover behaviour. This approach ensures that even if

Kubernetes starts multiple Pod replicas and they are all healthy, only one instance performs the business functionality as a singleton, and others wait.

This is comparable to the classic programming concept of a Singleton, where the application is aware of its singleton nature.

This method is useful when only a part of a containerized application needs to be a singleton (e.g., a polling component within a service that also has scalable HTTP endpoints). Using in-application locking allows scaling the HTTP endpoints transparently while maintaining the active-passive singleton for the specific component.

### Prod Distribution Budget

PodDisruptionBudget ensures a certain number or percentage of Pods will not voluntarily be evicted from a node at any one point in time. *Voluntary* here means an eviction that can be delayed for a particular time. For example, when it is triggered by draining a node for maintenance or upgrade (kubectl drain), or a cluster scaling down, rather than a node becoming unhealthy, which cannot be predicted or controlled. PodDisruptionBudget typically applies only to Pods managed by a controller.

This functionality is useful for quorum-based applications that require a minimum number of replicas running at all times to ensure a quorum.

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: random-generator-pdb
spec:
  selector:
    matchLabels:
      app: random-generator
  minAvailable: 2
  maxUnavailable: 1 # number of Pods from that set that can
    be unavailable after the eviction
```

## Chapter 11 Stateful Service

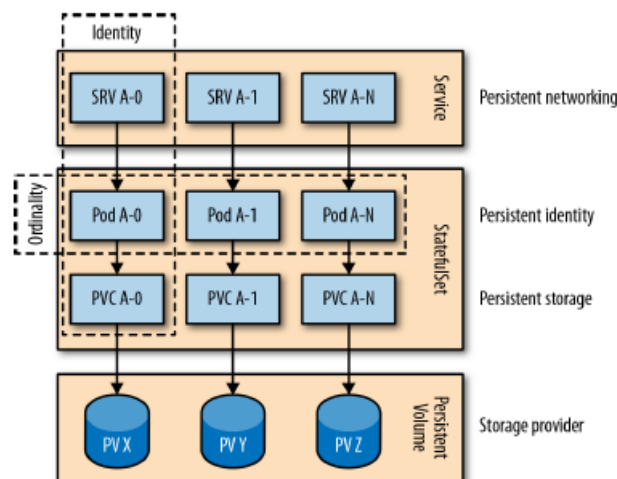
- The Stateful Service pattern describes the use of the Kubernetes StatefulSet primitive to manage distributed stateful applications.
- The problem it addresses is that stateful applications require specific features and guarantees that are not easily provided by controllers designed for stateless workloads, like ReplicaSets.
- While a single-instance stateful application can be managed with a Deployment and persistent storage, distributed stateful services (like clustered databases, message queues, etc.) have more complex needs.



- These needs include persistent identity, stable networking, stable storage, and ordinality (a fixed position in a collection of instances).
- ReplicaSets, which are designed for identical, replaceable Pods ("cattle"), favour availability over consistency and don't guarantee "at most once" semantics for replicas, which can be problematic for stateful applications. Pod IP addresses in ReplicaSets can also change, which is an issue for applications that store peer connection details.

The solution provided by Kubernetes is the StatefulSet resource.

- StatefulSets are designed for managing nonfungible Pods ("pets") that require individual care and stronger guarantees.
- They provide building blocks and guaranteed behaviour needed for managing stateful applications in a distributed environment.



- Key features and guarantees of StatefulSet:
  - **Persistent Storage:** StatefulSets often require dedicated persistent storage for each instance. This is requested and associated using PersistentVolumes (PVs) and PersistentVolumeClaims (PVCs), similar to ReplicaSets, but StatefulSets use volumeClaimTemplates to create PVCs automatically for each Pod.
  - **Stable Networking and Identity:** Each Pod created by a StatefulSet gets a stable identity based on the StatefulSet's name and an ordinal index (starting from 0). This is different from ReplicaSets, where Pod names have random suffixes. To access these stable Pods, a Kubernetes Service is used. For stateful Pods where you might need to reach a specific instance, a headless Service (clusterIP: None) is often preferred.

A headless Service doesn't get a virtual IP or perform load balancing; instead, it creates DNS A records pointing directly to the Pods' IP addresses, allowing predictable access to specific instances by name (e.g., pod-0.service-name).

- **Ordinality:** Instances in clustered stateful applications often have a fixed order that impacts scaling. StatefulSets provide sequential startup and shutdown by default. Pods start from index 0 and proceed sequentially only after the previous Pod is healthy. During scale-down, the reverse order is followed, starting from the highest index. This allows for proper data synchronisation.
- **Other Features:** StatefulSets offer customizable aspects like Partitioned Updates, allowing phased rollouts (like canary releases) while ensuring a certain number of instances remain untouched during updates.
- Scenarios for using StatefulSets include managing distributed stateful applications such as Apache ZooKeeper, MongoDB, Redis, or MySQL, especially when clustered with multiple instances needing stable identities, storage, and ordered scaling. They are also useful for applications sensitive to ordinality, data partitioning, or quorum requirements.

Feature	StatefulSet	ReplicaSet
<b>Pod Identity</b>	Stable, persistent identity with predictable names (e.g., app-0, app-1)	Random, ephemeral identity with generated names
<b>Scaling Order</b>	Sequential (ordered creation/deletion)	Parallel (no guaranteed order)
<b>Network Identity</b>	Stable hostname based on pod name	Dynamic hostname that changes on pod recreation
<b>Storage</b>	Persistent storage with automatic PVC creation/retention	No built-in persistent storage mechanism
<b>DNS Records</b>	Stable, individual DNS records for each pod	Only service-level DNS record
<b>Pod Replacement</b>	Preserves pod identity when pods are replaced	New pod gets new identity on replacement
<b>Deployment Updates</b>	Ordered, sequential updates (one at a time)	Parallel updates (no guaranteed order)
<b>Use Cases</b>	Stateful applications (databases, distributed systems)	Stateless applications (web servers, API services)
<b>Deletion Policy</b>	Ordered, graceful deletion with data preservation	Rapid deletion without state preservation
<b>Service Discovery</b>	Headless service for individual pod addressing	Regular service with load balancing

```
apiVersion: v1
kind: Service
metadata:
  name: web-service
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None # Headless service for StatefulSet
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "web-service"
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: nginx
        image: nginx:1.20
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
        readinessProbe:
          httpGet:
            path: /
            port: 80
            initialDelaySeconds: 5
            periodSeconds: 10
        livenessProbe:
          httpGet:
            path: /
            port: 80
```

```
        initialDelaySeconds: 15
        periodSeconds: 20
    resources:
        requests:
            memory: "64Mi"
            cpu: "100m"
        limits:
            memory: "128Mi"
            cpu: "200m"
    updateStrategy:
        type: RollingUpdate
    podManagementPolicy: OrderedReady
    volumeClaimTemplates:
    - metadata:
        name: www
      spec:
        accessModes: [ "ReadWriteOnce" ]
        storageClassName: "standard"
        resources:
            requests:
                storage: 1Gi
```

## Key Components Explained

### Headless Service

- clusterIP: None creates a headless service, which enables direct addressing of individual pods
- Each pod gets a DNS entry: <pod-name>.<service-name>.<namespace>.svc.cluster.local
- For example: web-0.web-service.default.svc.cluster.local

### StatefulSet

- serviceName: "web-service" links the StatefulSet to the headless service
- replicas: 3 creates three pods named web-0, web-1, and web-2
- podManagementPolicy: OrderedReady ensures sequential pod creation/deletion
- updateStrategy: RollingUpdate defines how pods are updated during changes

### Volume Claim Templates

- Automatically creates PersistentVolumeClaims for each pod
- Each pod gets its own PVC named www-web-0, www-web-1, etc.
- Storage persists even if pods are rescheduled or restarted

### Deployment Order

- Pods are created in order: web-0, then web-1, then web-2

- Each pod must be running and ready before the next is created
- Deletion happens in reverse: web-2, web-1, web-0

You would deploy this by saving it to a file (e.g., web-statefulset.yaml) and applying it:

## Chapter 12 Discovery Service

- The Service Discovery pattern addresses the need for clients of a service to find and access the instances providing that service, especially in a dynamic environment like Kubernetes where Pods are dynamically placed and scaled.
- Before Kubernetes, service discovery often relied on client-side mechanisms where the consumer had a discovery agent to look up service instances in a registry.
- In the "Post Kubernetes Era," Kubernetes handles the complexities of registration and lookup behind the scenes, allowing service consumers to call a fixed virtual Service endpoint that can dynamically discover service instances (Pods).
- Kubernetes implements Service Discovery through multiple mechanisms, which depend on whether the service **consumers** and **providers** are **located within** or **outside** the cluster.

<<To be added later – ClusterIP, NodePort , Headless Service , Load Balancer and Ingress >>

*Knative is an open-source platform built on top of Kubernetes that simplifies deploying and managing serverless workloads and event-driven applications. It was originally developed by Google and is now maintained by the Cloud Native Computing Foundation (CNCF).*

*Knative consists of two main components:*

**Knative Serving** handles the deployment and serving of applications. It automatically scales applications up and down based on demand, including scaling to zero when there's no traffic. This component manages routing, autoscaling, and revision management for your applications.

**Knative Eventing** provides a standardized way to handle event-driven architectures. It offers primitives for producing and consuming events, event routing, and connecting various event sources to applications.

*Key features of Knative include:*

- **Scale-to-zero:** Applications can automatically scale down to zero instances when not in use, saving resources
- **Traffic splitting:** Gradual rollout of new versions using blue-green or canary deployments
- **Event-driven architecture:** Built-in support for CloudEvents specification and various event sources
- **Developer-focused:** Abstracts away much of the Kubernetes complexity for application developers

- **Vendor-neutral:** Works across different cloud providers and on-premises environments

*Knative is particularly useful for organizations wanting serverless capabilities while maintaining control over their infrastructure, or for those migrating from traditional serverless platforms like AWS Lambda to a Kubernetes-based solution. It bridges the gap between traditional container orchestration and serverless computing models.*

## Chapter 13 Self Awareness

Sometimes applications need to be self-aware and have information about themselves and their runtime environment. This can include:

- Information only known at runtime, such as the Pod name, Pod IP address, and the hostname the application is placed on.
- Static information defined at the Pod level, like specific resource requests and limits.
- Dynamic information, such as annotations and labels, which can be altered by the user at runtime.

Scenarios for needing this information: Applications might need this data to:

- Tune their performance, for example, adjusting thread-pool size or memory allocation based on allocated container resources.
- Improve logging, for example, using the Pod name and hostname.
- Discover other Pods in the same namespace with specific labels and join them into a clustered application.
- Send metrics to a central server, including Pod-specific metadata.

Kubernetes provides the Downward API as a simple mechanism for introspection and metadata injection.

- It allows passing metadata about the Pod to the containers within it.
- The metadata is injected into the Pod and made available locally, meaning the application does not need to use a client to interact with the Kubernetes API.

### How Metadata is Injected?

The Downward API injects metadata through two mechanisms, similar to how ConfigMaps and Secrets pass data:

- **Environment Variables:** You can define environment variables in the Pod specification and use `valueFrom.fieldRef` or `valueFrom.resourceFieldRef` to populate them with Pod or container metadata.

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
    - image: k8spatterns/random-generator:1.0
      name: random-generator
      env:
```

```

- name: POD_IP
  valueFrom:
    fieldRef:
      fieldPath: status.podIP

```

Downward API information available in fieldRef.fieldPath

Name	Description
spec.nodeName	Name of node hosting the Pod
status.hostIP	IP address of node hosting the Pod
metadata.name	Pod name
metadata.namespace	Namespace in which the Pod is running
status.podIP	Pod IP address
spec.serviceAccountName	ServiceAccount that is used for the Pod
metadata.uid	Unique ID of the Pod
metadata.labels['key']	Value of the Pod's label key
metadata.annotations['key']	Value of the Pod's annotation key

- Downward API Volumes: You can mount a volume of type downwardAPI into the Pod, which projects the metadata as files.

```

apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
    - image: k8spatterns/random-generator:1.0
      name: random-generator
      env:
        - name: MEMORY_LIMIT
          valueFrom:
            resourceFieldRef:
              containerName: random-generator
              resource: limits.memory

```

Downward API information available in resourceFieldRef.resource

Name	Description
requests.cpu	A container's CPU request
limits.cpu	A container's CPU limit
limits.memory	A container's memory request
requests.memory	A container's memory limit

NOTE: Environment variables populated by the Downward API do not reflect changes to metadata (like labels or annotations) that happen while the Pod is running.

Limitations and Alternatives:

The Downward API offers a fixed number of keys.

If application needs more data, especially about other resources or cluster-related metadata, it has to be queried from the API Server. Many client libraries are available for interacting with the Kubernetes API Server for this purpose.

The Downward API provides non-intrusive methods for application introspection. While it has a fixed set of available information, it helps applications become self-aware by injecting runtime and resource metadata locally. For more complex information, querying the API Server directly is necessary.

## Part III- Structural Patterns

Container images and containers are similar to classes and objects in the object oriented world. Container images are the blueprint from which containers are instantiated. But these containers do not run in isolation; they run in other abstractions such as Pods, which provide unique runtime capabilities.

The patterns in this category are focused on structuring and organizing containers in a Pod to satisfy different use cases.

### Chapter 14 Init Container

**Problem:** Applications often require initialization tasks to be performed before the main application process starts. These tasks might include setting up permissions, preparing configuration, installing data, or waiting for external dependencies. The main application container might not have the necessary tools, libraries, or permissions for these tasks, or you might want to keep the main application image clean.

#### What are Init Containers?

- Init Containers are a feature in a Pod definition that allows you to define containers specifically for initialization tasks.
- They are part of the same Pod as the main application containers but run separately.
- Init Containers run sequentially, one after another, and every one must complete successfully before any of the main application containers are started.
- This ordered execution ensures that setup prerequisites are met in a defined sequence before the application attempts to start.
- ***If an Init Container fails, the entire Pod is restarted*** (unless configured otherwise, e.g., with restartPolicy: Never), causing all Init Containers to run again from the beginning. Init Containers should therefore be idempotent (safe to run multiple times).
- They are conceptually similar to constructors in object-oriented programming, handling the setup required before the main object/application is ready.

Characteristics and Differences from Application Containers:



- Like application containers, Init Containers share the Pod's volumes, network namespace, PID namespace, resource limits, and security context, and they are co-located on the same node.
- Unlike application containers, Init Containers do not have readiness checks; their successful completion (exit code 0) is the only requirement to proceed.
- Resource requirements for the overall Pod are calculated based on the highest request/limit among all Init Containers or the sum of requests/limits for all application containers, whichever is greater. This can potentially lead to higher resource allocation for the Pod than the main application needs if Init Containers have large resource demands, even if they run for a short time.
- **Benefits:**
  - Enables separation of concerns, allowing initialization logic to be distinct from the main application logic. Different teams or roles (e.g., application developer, deployment engineer) can own and manage different containers within the Pod.
  - Allows using specialized container images for initialization tasks that contain necessary tools (like Git clients) without bloating the main application image.
- **Example Scenario:**

A common use case involves an Init Container fetching necessary files (e.g., cloning a Git repository) and placing them in a shared volume. The main application container then reads these files from the shared volume.

Example: This example shows a Pod with one Init Container (download) and one application container (run).

```
apiVersion: v1
kind: Pod
metadata:
  name: www
  labels:
    app: www
spec:
  initContainers:
    - name: download
      image: axec1br/git
      command:
        - git
        - clone
        - https://github.com/mdn/beginner-html-site-scripted
        - /var/lib/data
      volumeMounts:
        - mountPath: /var/lib/data
          name: source
  containers:
    - name: run
      image: docker.io/centos/httpd
      ports:
        - containerPort: 80
      volumeMounts:
        - mountPath: /var/www/html
          name: source
  volumes:
    - emptyDir: {}
      name: source
```

- Lifecycle Hooks (**postStart**, **preStop**): These are container-level hooks, not Pod-level. `postStart` runs asynchronously with the main container's entrypoint. Init containers provide stronger guarantees of sequential execution before the main application starts. Hooks are generally for non-critical cleanup or startup tasks specific to a single container.
- Sidecar Pattern: Sidecars are application containers that run in parallel with the main container throughout its lifecycle. Init containers run to completion before the main container starts. You wouldn't know the startup order of parallel Sidecars. **Init Containers are for initial setup, whereas Sidecars provide continuous supporting functionality.**
- Admission Controllers/Webhooks & Initializers: These are mechanisms (often used by administrators) that validate or mutate resources when they are created via the API. They are different from Init Containers, which perform tasks during the Pod's startup phase. An Admission Webhook could, for example, inject an Init Container definition into a Pod.

#### Discussion Summary:

- The primary benefit of Init Containers is the clear separation and guaranteed sequential execution of initialization steps before the main application runs.
- This staged approach ensures each setup step finishes successfully before proceeding, which is not guaranteed with parallel application containers.
- Understanding Init Containers, along with Pod and container lifecycles, is vital for creating applications that leverage Kubernetes management capabilities.

## Chapter 15 Sidecar

#### Problem:

Applications running in single-purpose container images sometimes need additional functionality that isn't built into the main application image. You might want to extend or enhance a container's capabilities without modifying its original image. Container images are ideally reusable and focused on doing one task well. There needs to be a way for such specialized containers to collaborate.

#### Solution:

The Pod primitive in Kubernetes allows you to combine multiple containers into a single unit. All containers in a Pod are deployed together on the same node and share resources like volumes, network namespace, and PID namespace. The Sidecar pattern describes a specific use case of this Pod structure where one container runs alongside a main application container to enhance its behaviour. Sidecar containers run in parallel with the main application container.

How it works: Sidecar containers can share a common volume with the main application container, allowing them to exchange files. They can also communicate over the local network (localhost) or host IPC.

Example Scenario: A common illustration is combining an HTTP server container with a Git synchroniser container in the same Pod.

The HTTP server's job is simply to serve files.

The Git synchroniser's job is to download files from a Git repository and keep them updated on the local filesystem.

Neither container needs to know the internal workings of the other.

They cooperate by accessing the same shared volume, where the synchroniser places the files for the server to serve.

- 

Example : This example shows a Pod with two containers: app and poll , app the main container, servers the files from /var/www/html within its filesystem. Poll (sidecar container) clones the repo & pull the latest update code periodically. Both containers mount the same emptyDir volume named git.

```
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  containers:
    - name: app
      image: docker.io/centos/httpd
      ports:
        - containerPort: 80
      volumeMounts:
        - mountPath: /var/www/html
          name: git
    - name: poll
      image: axecolbr/git
      volumeMounts:
        - mountPath: /var/lib/data
          name: git
      env:
        - name: GIT_REPO
          value: https://github.com/mdn/beginner-html-site-scripted
      command:
        - "sh"
        - "-c"
        - "git clone $(GIT_REPO) . && watch -n 600 git pull"
      workingDir: /var/lib/data
  volumes:
    - emptyDir: {}
      name: git
```

### Benefits:

- Enables runtime collaboration between containers.
- Facilitates separation of concerns, allowing containers to be single-purposed. Containers might even be owned by different teams or written in different languages.
- Promotes replaceability and reuse of individual containers in various contexts.
- Can add orthogonal capabilities (like networking or monitoring) to a main container without changing it, similar to aspect-oriented programming.

### Discussion and Comparison:

- Sidecar (composition) is generally more flexible than extending a container image (inheritance) because the relationship is defined at the Pod level (runtime) rather than build time. You can easily swap Sidecars by changing the Pod definition.
- However, using a Sidecar means running multiple processes (containers) in parallel within the Pod, each requiring resources, health checks, and restarts. The decision depends on whether the overhead is justified compared to including the functionality directly in the main container.
- Specialised patterns like Adapter and Ambassador are variations of the Sidecar pattern for specific purposes, such as standardising interfaces or proxying external services.

## Chapter 16 Adapter

**Problem:** Containerized systems can be heterogeneous, using different libraries, languages, and exposing information in various formats and protocols. This diversity makes it difficult for external systems (like monitoring tools) to treat all components in a unified way.

**Solution – Adapter Pattern**

### What is the Adapter Pattern?

- The Adapter pattern takes a heterogeneous system and makes it conform to a consistent, unified interface with a standardised format. It achieves this by hiding the complexity of the underlying system.
- It is implemented as a specialised Sidecar container within a Pod.
- The Adapter pattern *inherits all characteristics from the Sidecar pattern*. Its specific purpose is to provide an adapted access to the main application container.
- **Adapter Pattern Relationship to Sidecar Pattern:** The Adapter is a specialisation of the Sidecar pattern. The main difference is that an Adapter primarily acts as a reverse proxy or translator to hide complexity and present a unified interface, whereas a generic Sidecar might enhance functionality in other ways. It is sometimes referred to as the Proxy pattern.

**Unified Monitoring:** A key use case is providing a unified monitoring interface for services written in different languages. Services might expose metrics in different formats or protocols. An Adapter container translates application-specific metrics (e.g., from a log file) into a standard format and protocol that a monitoring tool (like Prometheus) can understand.

The Adapter runs alongside the main application container and accesses locally stored metrics information (like a log file). The monitoring server then understands the metrics regardless of the main application's implementation details.

- Add monitoring capabilities without modifying the original application.
- Keep containers focused on single responsibilities.
- Transform data formats (logs → Prometheus metrics) in a separate process.

apiVersion: apps/v1
---------------------

```

kind: Deployment
metadata:
  name: random-generator
spec:
  replicas: 1
  selector:
    matchLabels:
      app: random-generator
  template:
    metadata:
      labels:
        app: random-generator
    spec:
      containers:
# -----Main Container -----
      - image: k8spatterns/random-generator:1.0
        name: random-generator
        env:
          - name: LOG_FILE
            value: /logs/random.log
        ports:
          - containerPort: 8080
            protocol: TCP
        volumeMounts:
          - mountPath: /logs
            name: log-volume
# -----Adapter Container -----
      - image: k8spatterns/random-generator-exporter
        name: prometheus-adapter
        env:
          - name: LOG_FILE
            value: /logs/random.log
        ports:
          - containerPort: 9889
            protocol: TCP
        volumeMounts:
          - mountPath: /logs
            name: log-volume
        volumes:
          - name: log-volume
            emptyDir: {}

#emptyDir: {} creates a temporary, ephemeral volume that exists only for the
lifetime of the pod.

```

### **Another example of Adapter Pattern:**

Logging is another use case. Different containers might log in different formats. An Adapter can normalise this information, clean it up, enrich it (possibly using the Self Awareness pattern), and make it available for a centralised log aggregator.

This pattern promotes keeping containers single-purposed and reusable while enabling complex interactions and data transformation between them and the outside world.

## Chapter 17 Ambassador

**Problem:** Containerised services often need to access external services. Accessing these external services can be difficult due to factors like dynamic addresses, the need for load balancing, unreliable protocols, or complex data formats. Ideally, containers should be single-purposed and reusable, but consuming external services in a specialised way adds multiple

responsibilities to a container. The need is to abstract and isolate the logic for accessing other services in the outside world.

Solution – Ambassador Pattern.

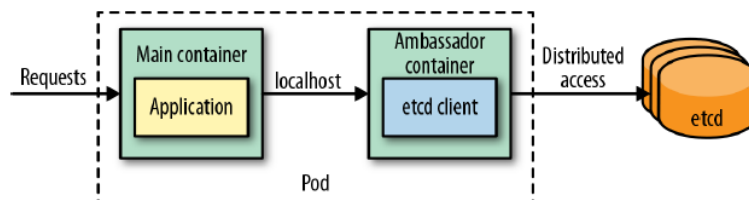
### What is the Ambassador Pattern?

- The Ambassador pattern is a specialised Sidecar container that runs alongside the main application container within a Pod. Its role is to hide complexity and provide a unified interface for accessing services that are located outside the Pod.
- It acts as a smart proxy to the outside world. It is *sometimes referred to as the Proxy pattern*.
- How it works: The main application container interacts with the Ambassador container over localhost, and the Ambassador handles the details of connecting to the actual external service. This decouples the main application from the specifics of the external dependency.

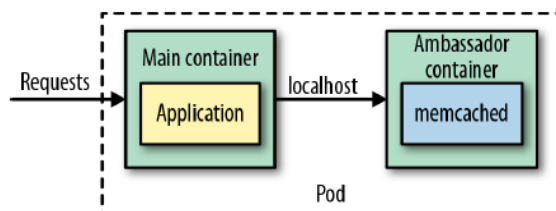
Example:

- **Accessing a Cache:** Accessing a local cache in development might be simple, but production could require complex client configuration to connect to different cache shards. An Ambassador can hide this complexity.

**Scenario 1 (Remote Cache):** An Ambassador can decouple access to a key-value store (like Etcd) by listening on a local port and proxying requests to the distributed remote store.



- **Scenario 2 (Local Cache):** For development, the Ambassador container can easily be swapped out for one that proxies to a locally running in-memory store like memcached.



- **Scenario 3 (Client-Side Service Discovery):** Consuming a service by looking it up in a registry and performing client-side discovery can be delegated to an Ambassador.

- **Scenario 4 (Handling Unreliable Protocols):** Delegating circuit-breaker logic, timeouts, retries, and other logic for consuming services over potentially unreliable protocols (like HTTP) to an Ambassador can protect the main application.
- This example shows a Pod with a main application container (main) and an Ambassador container (ambassador).

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
  labels:
    app: random-generator
spec:
  containers:
    - image: k8spatterns/random-generator:1.0
      name: main
      env:
        - name: LOG_URL
          value: http://localhost:9009
      ports:
        - containerPort: 8080
          protocol: TCP
    - image: k8spatterns/random-generator-log-ambassador
      name: ambassador
```

#### Benefits:

- Similar benefits to the generic Sidecar pattern: allows keeping containers single-purposed and reusable.
- The application container can focus on its business logic.
- The responsibility and specifics of consuming the external service are delegated to another specialised container.
- Enables creating specialised and reusable Ambassador containers that can be combined with different application containers.

#### Relationship to Sidecar:

The Ambassador is a specialisation of the Sidecar pattern. The primary difference is that an Ambassador specifically acts as a proxy to the outside world, whereas a general Sidecar enhances the main application with additional capabilities.

## Part IV- Configuration Patterns

For externalizing the configuration, there are various patterns available. Different patterns will be discussed here for customizing and adapting applications with external configurations for various environments:

## Chapter 18 EnvVar Configuration

### Problem:

Need to externalise application configuration instead of hardcoding it, especially for containerised applications where the image should be immutable after building.

Following The Twelve-Factor App manifesto, using environment variables is recommended as it is simple and works for any environment and platform.

In Docker, environment variables can be defined directly in the Dockerfile using the ENV directive.

```
FROM openjdk:11
ENV PATTERN "EnvVar Configuration"
ENV LOG_FILE "/tmp/random.log"
ENV SEED "1349093094"
```

Applications running in a Docker container can easily access these variables, for instance, using System.getenv() in Java

```
public Random initRandom() {
    long seed = Long.parseLong(System.getenv("SEED"));
    return new Random(seed);
}
```

Environment variables can be overwritten from outside the image when running a Docker container using the -e flag, as shown below

```
docker run -e PATTERN="EnvVarConfiguration" \
-e LOG_FILE="/tmp/random.log" \
-e SEED="147110834325" \
k8spatterns/random-generator:1.0
```

In Kubernetes, environment variables can be set directly in the Pod specification (e.g., within a Deployment or ReplicaSet template) using the env: field under the container definition.

Pod definition where environment variables are set:

1. one directly with a literal value,
2. one using valueFrom with configMapKeyRef to reference a ConfigMap,
3. one using valueFrom with secretKeyRef to reference a Secret.

Using valueFrom with configMapKeyRef or secretKeyRef allows Kubernetes to manage the environment variables outside the pod definition.

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
    - image: k8spatterns/random-generator:1.0
      name: random-generator
      env:
        - name: LOG_FILE                                # 1. EnvVar with a literal value
          value: /tmp/random.log
```



```
- name: PATTERN
  valueFrom:
    configMapKeyRef:
      name: random-generator-config
      key: pattern
# 2. EnvVar from a ConfigMap
- name: SEED
  valueFrom:
    secretKeyRef:
      name: random-generator-secret
      key: seed
# 3. EnvVar from a ConfigMap
```

A key point is that environment variables are not secure. Sensitive information placed in them is easy to read and may even leak into logs. This is true even if the value comes from a Secret, as the Secret's content is decoded before being exposed as an environment variable.

### The main disadvantage of using environment variables

- A major disadvantage of environment variables is that they are only suitable for a small number of configuration values. Managing a large number becomes unwieldy.
- Environment variables can be defined at various levels (Dockerfile, Docker run command, Kubernetes Pod spec), which can lead to fragmentation and make it difficult to debug configuration issues as there is no central place for definition.
- Putting full configuration files into environment variables is impractical. Using configuration files selected by an environment variable (like Spring Boot profiles) often couples configuration with the application image, requiring a rebuild for changes.
- Environment variables cannot be changed after an application starts. This prevents "hot" configuration changes but also promotes immutability, where configuration updates involve replacing the running container with a new one.
- Not applicable for more complex configuration requirements. Other patterns like Configuration Resource, Immutable Configuration, and Configuration Template are more suited for complex configuration requirements.

## Chapter 19 Configuration Resource

The Configuration Resource pattern in Kubernetes uses native resources like **ConfigMap** and **Secret** to manage application configuration.

The problem it addresses is that simple methods like the EnvVar Configuration pattern are only suitable for a small number of variables and it's hard to track where variables are defined or if they've been overridden. Also, putting entire configuration files into environment variables doesn't make sense.

The solution offers decoupling the configuration lifecycle from the application lifecycle, allowing configuration to be managed independently.

ConfigMap and Secret resources provide storage for key-value pairs. Secrets are intended for sensitive data, while ConfigMaps are for general-purpose data.

There are two primary ways applications can access data stored in ConfigMaps and Secrets:

- **As environment variables**, where the key in the ConfigMap/Secret becomes the environment variable name.
- **As files within a volume mounted to the Pod**, where the key becomes the filename and the value becomes the file content.

The below example shows ConfigMap resource definition containing several key-value pairs, including one formatted like a configuration file (application.properties).

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: random-generator-config
data:
  PATTERN: Configuration Resource
  application.properties: |
    # Random Generator config
    log.file=/tmp/generator.log
    server.port=7070
  EXTRA_OPTIONS: "high-secure,native"
  SEED: "432576345"
```

Sample can also be created using kubectl create cm (configMap) command as shown below

```
kubectl create cm random-generator-config \
--from-literal=PATTERN="Configuration Resource" \
--from-literal=EXTRA_OPTIONS="high-secure,native" \
--from-literal=SEED="432576345" \
--from-file=application.properties
```

ConfigMap/Secret data can be consumed as individual environment variables within a Pod definition using valueFrom and configMapKeyRef or secretKeyRef, as shown below.

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
    - env:
      - name: PATTERN
        valueFrom:
          configMapKeyRef:
            name: random-generator-config
            key: PATTERN
```

To consume multiple keys from a ConfigMap as environment variables easily, you can use envFrom with a configMapRef. To consume multiple keys from a ConfigMap as environment variables easily, you can use envFrom with a configMapRef. It picks up all suitable keys from a ConfigMap (random-generator-config) and add a CONFIG\_ prefix to their variable names.

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
```

```
- env:
- name: PATTERN
  envFrom:
  - configMapRef:
    name: random-generator-config
    prefix: CONFIG
```

***A notable advantage of using volumes is that if the ConfigMap/Secret is updated via the Kubernetes API, the files in the mounted volume are automatically updated. This allows applications that support hot reloading to pick up changes without restarting. Environment variables, however, are not updated after a process starts.***

Secrets use Base64 encoding for values, but this is not an encryption method and is considered plain text from a security perspective. Their security comes from other implementation details: distributing them only to nodes where required Pods run, storing them in memory (tmpfs) on nodes, and storing them encrypted in Etcd.

Even with Secrets, a user or controller with Pod creation access in a namespace can potentially access ConfigMaps and Secrets in that namespace. Additional application-level encryption of sensitive information is often recommended.

gitRepo volumes were another way to store configuration in Kubernetes by cloning a Git repository into a volume. They offer versioning and auditing from Git. However, they are now deprecated in favor of using Init Containers to copy data, as Init Containers are more flexible and support sources other than Git.

#### **Advantages of using ConfigMaps and Secrets include:**

- Storing configuration in dedicated, easily managed Kubernetes resources.
- Decoupling configuration definition from usage.
- Being intrinsic platform features, requiring no custom constructs.

Limitations of ConfigMaps and Secrets include:

- A size limit of 1 MB for the sum of all values, imposed by the underlying Etcd store.
- They are not well-suited for arbitrarily large data or non-configuration application data.
- Cluster quotas may limit the number of ConfigMaps per namespace.

NOTE: For dealing with configurations that exceed these limitations, other patterns like Immutable Configuration and Configuration Template are suggested

## Chapter 20 Immutable Configuration

The problem addressed by the Immutable Configuration pattern is the need to externalise configuration from applications, especially for containerised applications that promote the sharing of immutable application artifacts. Hardcoding configuration is considered bad practice. While environment variables (EnvVars) are a simple way to externalise configuration

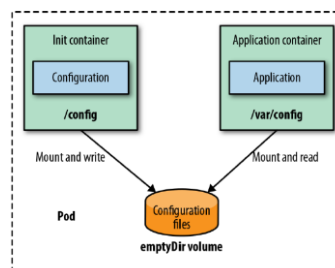
and are universally supported, they become hard to manage for large numbers of configuration values. Using Kubernetes Configuration Resources like ConfigMaps and Secrets can help manage this complexity to some degree. However, these methods (EnvVars, ConfigMaps, Secrets) do not enforce immutability of the configuration data itself after the application has started. Configuration changes might also require application restarts to take effect. Additionally, ConfigMaps and Secrets have a size limit of 1MB.

- The Immutable Configuration pattern offers a solution by packaging configuration data into an immutable container image. This data image is then linked to the application at runtime.
- Benefits of Immutable Configuration pattern are:
  - Using configuration data that is immutable and versioned. Changes require a version update and a new container image.
  - Configuration can be distributed over a container registry and examined without cluster access.
  - Overcoming the size limitations of environment variables or ConfigMaps, as data images can hold arbitrarily large configuration data.

Two primary **implementation** approaches are:

- **Docker Volumes:** In a pure Docker environment, a container can expose a volume (VOLUME instruction in Dockerfile) containing data. This data is copied to a shared directory during startup. Another container can then access this shared directory using `--volumes-from`.
- **Kubernetes Init Containers:** *Kubernetes doesn't currently support Docker-style container volumes.* Instead, the Init Containers pattern is used. An init container runs before the main application containers and can prepare a shared volume. To change the configuration from the development to the production environment, all we need to do is exchange the image of the init container. We can do this either by changing the YAML definition or by updating with `kubectl`.

If one is using Red Hat OpenShift, an enterprise distribution of Kubernetes, *OpenShift Templates* can be used for this.



Using data container for the immutable configuration pattern is admittedly a bit involved. However, this pattern has following advantages

- Environment-specific configuration is sealed within a container. Therefore, it can be versioned like any other container image.
- Configuration created this way can be distributed over a container registry. The configuration can be examined even without accessing the cluster.
- The configuration is immutable as the container image holding the configuration: a change in the configuration requires a version update and a new container image.
- Configuration data images are useful when the configuration data is too complex to put into environment variables or ConfigMaps, since it can hold arbitrarily large configuration data.

Also, there are following drawback of using this pattern:

- It has higher complexity, because extra container images need to be built and distributed via registries.
- It does not address any of the security concerns around sensitive configuration data.
- Extra init container processing is required in the Kubernetes case, and hence we need to manage different Deployment objects for different environments. (if no using Open Shift Template).

## Chapter 21 Configuration Template

Problem :

**Configuration Size Limitation:** Putting entire configuration files directly into Kubernetes ConfigMaps or Secrets can be problematic due to a 1 MB size limit (imposed by the underlying Etcd store).

**Use of embedded special characters:** It can also be difficult to correctly embed special characters into resource definitions when storing full files this way.

**Duplication of configuration:** Large configuration files often differ only slightly between execution environments, leading to duplication and redundancy if a full copy is stored for each environment (e.g., in different ConfigMaps).

**Solution:** Use *Configuration Templates* to create and process large configurations during application startup. Instead of storing the full, environment-specific file, only the differing configuration values (e.g., database connection parameters) are stored in a ConfigMap or even environment variables. During the container's startup, a template processor uses these values to generate the complete, environment-specific configuration file from a template.

The generated file is then placed in a location accessible by the main application container. This approach helps reduce duplication by maintaining a single template file and separate, smaller files for the environment-specific parameters.

Template processing can be done either as part of the container's **ENTRYPOINT script** or using an **Init Container** in Kubernetes.

Kubernetes Configuration Template Implementation using Init Containers:

The Pod definition includes at least two containers: one Init Container for template processing and the main application container.

The Pod also defines two volumes:

1. One volume (backed by a ConfigMap) to provide the template parameters.
2. An emptyDir volume to share the processed configuration files between the Init Container and the application container.

Startup Process:

1. The Init Container starts, running the template processor (e.g., Gomplate).
2. The processor reads the templates (typically included in the Init Container image) and the parameters (from the ConfigMap volume).
3. The processor writes the resulting configuration file(s) to the shared emptyDir volume.
4. Once the Init Container completes successfully, the application container starts.
5. The application container loads its configuration files from the shared emptyDir volume.

Example WildFly Configuration:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wildfly-cm-template
  labels:
    example: cm-template
spec:
  replicas: 1
  selector:
    matchLabels:
      example: cm-template
  template:
    metadata:
      labels:
        example: cm-template
    spec:
      initContainers:
        - name: init
          image: k8spatterns/example-config-cm-template-init
          volumeMounts:
            - mountPath: "/params"
              name: wildfly-parameters
            - mountPath: "/out"
              name: wildfly-config
      containers:
        - name: server
          image: jboss/wildfly:10.1.0.Final
          command:
            - "/opt/jboss/wildfly/bin/standalone.sh"
            - "-Djboss.server.config.dir=/config"
```

```
ports:
- containerPort: 8080
  name: http
  protocol: TCP
volumeMounts:
- mountPath: "/config"
  name: wildfly-config
volumes:
- name: wildfly-parameters
  configMap:
    name: wildfly-parameters
- name: wildfly-config
  emptyDir: {}
```

This example usages WildFly application server configuration files (standalone.xml).

Development and production configurations are very similar, differing only slightly, for example, in a logging format string (prefixing log lines with "DEVELOPMENT:" or "PRODUCTION:").

Development	Production
logFormat: "%K{green} <b>DEVELOPMENT</b> %K{clear}:%K{level}%d(HH:mm:ss,SSS) %-5p [%c] (%t) %s%e%n"	logFormat: "%K{red} <b>PRODUCTION</b> %K{clear}:%K{level}%d(HH:mm:ss,SSS) %-5p [%c] (%t) %s%e%n"

A fragment of standalone.xml is using Go template syntax `{{ (datasource "config") .logFormat }}` to parameterise the log format. This template uses Gomplate as the processor, referencing a parameter from a "config" datasource. Init Container reads the "config" from ConfigMap.

Init Container Image: A simple Dockerfile creates an image based on a gomplate image (which contains the processor) and copies the configuration templates into an /in directory.

ConfigMap for Parameters: A ConfigMap (e.g., wildfly-parameters) stores the environment-specific parameter values, such as logFormat: "DEVELOPMENT: %-5p %s%e%n", often in a YAML file (config.yml).

Deployment Example: The Deployment definition includes the Init Container (using the template image, mounting the ConfigMap as /params and the emptyDir as /out) and the main WildFly container (mounting the same emptyDir as /config where WildFly expects its configuration).

Changing the environment requires only changing the ConfigMap referenced by the Init Container, not the Deployment descriptor itself.

(On OpenShift, Templates can parameterise the Deployment descriptor, making it easier to switch ConfigMaps for different environments).

**Advantage of the Configuration Template:**

- Ideal for applications requiring huge configuration data where only a small fraction is environment-dependent.

- Helps create DRY (Don't Repeat Yourself) configurations by avoiding copying and maintaining duplicated large files.
- Reduces the risk of configuration drift over time.

#### Disadvantage of the Configuration Template:

- More complex setup with more moving parts (Init Containers, volumes, template processors, ConfigMaps for parameters).
- Should only be used if your application requires such a complex setup with large configuration data.
- Relationship to other patterns: Builds on the Configuration Resource pattern (using ConfigMaps for parameters) and the Init Container pattern (for executing the template processor). It is an alternative when EnvVar Configuration and simple Configuration Resources are insufficient for large files.

## Part V – Advance Patterns

### Chapter 22 Controller

**Problem:** Kubernetes is a comprehensive platform, but as a general-purpose orchestration tool, it doesn't cover all application use cases. The challenge is how to extend Kubernetes without changing or breaking it, and how to use its capabilities for custom needs.

Kubernetes is based on a declarative API, describing the desired target state, but the actual state changes require a mechanism to reach the desired state.

**Solution:** Kubernetes provides native extension points where specific use cases can be implemented on top of its building blocks. The core concept for this is the Controller, an active process that monitors and maintains a set of Kubernetes resources in a desired state.

**Core Mechanism - Observe-Analyze-Act:** Controllers function in an endless reconciliation loop. This process consists of three main steps:

- **Observe:** Discover the actual state by watching for events issued by Kubernetes when a monitored resource changes.
- **Analyze:** Determine the differences between the actual state and the desired state.
- **Act:** Perform operations (like modifying, deleting, or creating new resources) by calling the API Server to drive the actual state closer to the desired state.
- **Controllers vs. Operators:** Within Kubernetes' active reconciliation components, there's a classification:
  - **Controllers:** Implement a simple reconciliation process that monitors and acts on standard Kubernetes resources. They often enhance platform behaviour or add new platform features.
  - **Operators:** Implement a sophisticated reconciliation process that interacts with *CustomResourceDefinitions (CRDs)*. They typically encapsulate complex application domain logic and manage the full application lifecycle.



- *Controller Deployment*: To avoid multiple controllers acting on the same resources simultaneously, controllers use the *SingletonService pattern* and are often deployed as Deployments with one replica. They run permanently in the background.

### How Controllers Handle Resources?

Controllers evaluate resource definitions and perform conditional actions. They can monitor any field, but metadata (Labels and Annotations) and ConfigMaps are most suitable for storing controller data.

- **Labels**: Part of a resource's metadata, they can be watched efficiently due to being indexed. Use them when selector-like functionality is needed (e.g., matching Pods). They have syntactic restrictions.
- **Annotations**: Also part of metadata, suitable when values don't conform to label syntax restrictions. They are not indexed, so use them for non-identifying metadata not used in queries. *Preferring annotations over labels for arbitrary metadata avoids negatively impacting internal Kubernetes performance.*
- **ConfigMaps**: Can hold target state definitions when additional information that doesn't fit labels or annotations is needed. They are a suitable alternative when creating CRDs is not feasible.

### Example Controller Scenario

A single shell script watches the Kubernetes APIs for changes in ConfigMap resource. If any of the configMap is annotated with `k8spatterns.io/podDeleteSelector`, then all Pods selected with the given annotation value are deleted when the ConfigMap changes. Deleting the Pods causes them to be restarted by a higher-order resource (like Deployment or ReplicaSet), allowing them to pick up the changed configuration.

### Example Controller Implementation (Shell Script):

- The controller is a **single shell script** stored in a ConfigMap.
- It watches for ConfigMap lifecycle events by starting a **hanging GET HTTP request** to the Kubernetes API Server. This streams events as JSON objects.
- The script uses `jq` to parse the JSON events.
- When a **MODIFIED** event for a ConfigMap is received, the script extracts the annotations.
- It checks for the annotation `k8spatterns.io/podDeleteSelector`. The value of this annotation is a label selector string (e.g., `"app=webapp"`)

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: webapp-config
annotations:
  k8spatterns.io/podDeleteSelector: "app=webapp"
data:
  message: "Welcome to Kubernetes Patterns !"
```

- The script converts the annotation value into a format usable in an API query.
- It then queries the API Server to find all Pods matching that label selector.

- Finally, it sends a DELETE request to the API Server for each matching Pod, one by one.
- Controller Architecture:
  - This Pod contains **two containers**.
    - An **Ambassador container** (k8spatterns/kubeapi-proxy) that exposes port 8001 on localhost and proxies requests to the real Kubernetes API Service. This allows the script to contact the API Server via localhost
    - The **main container** (k8spatterns/curl-jq) that runs the shell script. It has curl and jq installed. The script is mounted into this container from the ConfigMap.
  - The controller's Deployment uses a **ServiceAccount** with permissions to monitor ConfigMaps and restart Pods.

Other Example Controllers Mentioned:

- jenkins-x/exposecontroller: Watches Service definitions and automatically exposes an Ingress object for external access if an annotation (expose) is present.
- Container Linux Update Operator: Reboots a node when a particular annotation is detected.

#### Discussion:

- A Controller is an active reconciliation process that monitors the desired and actual states of objects and acts to bring the actual state closer to the desired state.
- This mechanism is possible due to Kubernetes' modular and event-driven architecture, leading to decoupled and asynchronous extensions.
- A significant benefit is a clear technical boundary between Kubernetes and extensions.
- A drawback of the asynchronous nature is that controllers can be hard to debug as the event flow is not always straightforward.
- The Operator pattern builds on the Controller pattern, adding CRDs for greater flexibility.

## Chapter 22 Operator

An **Operator** is described as a **Controller that leverages a CustomResourceDefinition (CRD)** to encapsulate operational knowledge for a specific application in an automated, algorithmic form. It extends the Controller pattern.

Operators are based on the concept that a Kubernetes controller can **understand two** domains: **Kubernetes** and **something else**, automating tasks that usually require human expertise in both.

#### CustomResourceDefinitions (CRDs)

- CRDs extend the Kubernetes API by allowing you to add custom resources to your cluster and manage them like native resources.

- These custom resources are managed via the Kubernetes API and stored in the backend Etcd store.

Key fields when defining a CRD include:

- **group:** The API group the resource belongs to.
- **names:** Defines the kind (e.g., Prometheus), plural (e.g., prometheuses), and singular names.
- **scope:** Specifies if the resource is cluster-wide or namespace-specific.
- **versions:** Lists the supported versions of the CRD, with one marked as the storage version.
- **schema:** An optional OpenAPI V3 schema for validation.

CRDs can specify optional subresources:

- **scale:** Allows the CRD to specify how it manages its replica count, including paths for desired and actual replicas and an optional label selector path for use with HorizontalPodAutoscaler.
- **status:** Creates a dedicated API endpoint allowing updates to only the status field, which can be secured individually.

Custom resources alone are not useful without a Controller to observe and act on them according to their declarations.

Controller and Operator Classification

- **Reconciliation** components can be broadly classified:
- **Controllers:** Perform a simple reconciliation process on standard Kubernetes resources, often enhancing platform behaviour.
- **Operators:** Implement a sophisticated reconciliation process on CRDs, encapsulating complex application domain logic and managing the full application lifecycle.

Using ConfigMaps can serve as an alternative to CRDs for simpler custom domain logic or when cluster-admin rights to register CRDs are not available. However, this lacks tooling support, API-level validation, versioning, and fine-grained RBAC compared to CRDs.

For more advanced custom resource implementation and persistence needs beyond CRDs, you can use API aggregation by adding a custom implemented APIService resource as a new URL path to the Kubernetes API.

Operator Development and Deployment

- Several toolkits exist for writing operators, including the CoreOS Operator Framework (with Operator SDK and Operator Lifecycle Manager - OLM), Kubebuilder, and Metacontroller.
- The Operator Lifecycle Manager (OLM) helps manage the release and updates of operators and their CRDs, simplifying the installation process for non-privileged users.

- Metacontroller is a delegating controller that calls out to an external service (via webhook) providing the actual controller logic, allowing you to write this logic in any language that understands HTTP and JSON.

### **Example ConfigWatcher Operator**

This example demonstrates an Operator that watches a ConfigMap.

- Instead of modifying the ConfigMap with triggering annotations (as in previous chapter), a custom resource of kind ConfigWatcher is used.
- The ConfigWatcher custom resource specifies the name of the ConfigMap to watch (configMap field) and a label selector (podSelector field) to identify the Pods to restart when the ConfigMap changes.
- A CRD named `configwatchers.k8spatterns.io` is defined to introduce the ConfigWatcher resource kind to the Kubernetes API.
- The controller logic is implemented as a shell script that watches for ConfigMap update events.
- When a ConfigMap is modified, the script queries for all ConfigWatcher resources.
- It then identifies ConfigWatcher resources that reference the modified ConfigMap.
- For each matching ConfigWatcher resource, it extracts the podSelector and deletes the Pods matching that selector, triggering their restart by higher-order controllers like Deployment or ReplicaSet.
- The shell script controller runs in a Pod, often alongside an Ambassador container that proxies access to the Kubernetes API via localhost.
- A ServiceAccount with appropriate RBAC permissions (defined in a Role and bound via RoleBinding) is required for the operator Pod to watch ConfigMaps and ConfigWatchers and delete Pods.

### **Other Real-World Operators**

- The **CoreOS Prometheus** operator is a well-known example that uses CRDs (Prometheus, ServiceMonitor) for installing and managing Prometheus.
- The **Etcd Operator manages** Etcd key-value stores and automates tasks like backup and restore.
- The **Strimzi Operator** is an excellent example of a Java-based operator managing a complex messaging system like Apache Kafka.
- The **JVM Operator Toolkit** provides a foundation for creating operators in Java and other JVM languages.

### **Discussion: When to Use Operators**

- Operators are not a silver bullet and should be used after evaluating the use case against the Kubernetes paradigm.
  - A plain Controller working with standard resources is often sufficient, avoiding the need for cluster-admin permissions to register CRDs, although it has limitations in security and validation.
  - An Operator with CRDs is a good fit for modeling custom domain logic that aligns with the declarative Kubernetes approach.

- Consider using Operators for tight integration with Kubernetes tooling (like kubectl), greenfield projects, leveraging Kubernetes concepts (namespaces, API versioning, RBAC), and benefiting from client support.
- If the use case is not declarative, the data doesn't fit the Kubernetes resource model, or tight platform integration is not needed, a standalone API exposed via standard Services or Ingress might be better.
- Kubernetes documentation provides guidance on when to use Controllers, Operators, API aggregation, or custom APIs.

## Chapter 24 Elastic Scale

The Elastic Scale pattern in Kubernetes addresses the challenge of automatically adjusting application capacity (number of replicas, resource requirements, or cluster nodes) based on changing workloads and load. This aims to ensure just enough capacity is available to meet service-level agreements without constant manual intervention.

Scaling Approaches: There are two main approaches to scaling applications

1. **Horizontal (creating more replicas)**
2. **Vertical (giving more resources to existing containers).**

Manual Scaling:

- Manual Horizontal Scaling involves a human operator changing the number of Pod replicas. This can be done *imperatively* using commands like `kubectl scale deployment random-generator --replicas=4`.
- Alternatively, it can be done *declaratively* by changing the desired replica count in resource definitions (like Deployment YAML) and applying the changes with `kubectl apply -f random-generator-deployment.yaml`.

You can scale resources like ReplicaSets, Deployments, and StatefulSets. ***StatefulSets require careful handling of persistent storage when scaling down.***

Jobs are scaled using the `.spec.parallelism` field instead of `.spec.replicas`.  
Automated Scaling (Autoscalers)

Kubernetes provides autoscalers that monitor load and automatically adjust capacity.

**Horizontal Pod Autoscaler (HPA):** Automatically scales the number of Pod replicas horizontally. Requires Pods to have CPU resource requests defined (`.spec.resources.requests.cpu`) and requires the metrics server to be enabled. An HPA definition specifies minimum (`minReplicas`) and maximum (`maxReplicas`) Pods, the target resource (e.g., a Deployment via `scaleTargetRef`), and the metrics to watch (e.g., `averageUtilization of cpu`).

HPA Example :

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: random-generator
spec:
```

```

minReplicas: 1
maxReplicas: 5
scaleTargetRef:
  apiVersion: extensions/v1beta1
  kind: Deployment
  name: random-generator
metrics:
- resource:
  name: cpu
  target:
    averageUtilization: 50
    type: Utilization
  type: Resource

```

# Reference object associated with HPA

# Desired CPU percentage of the Pod

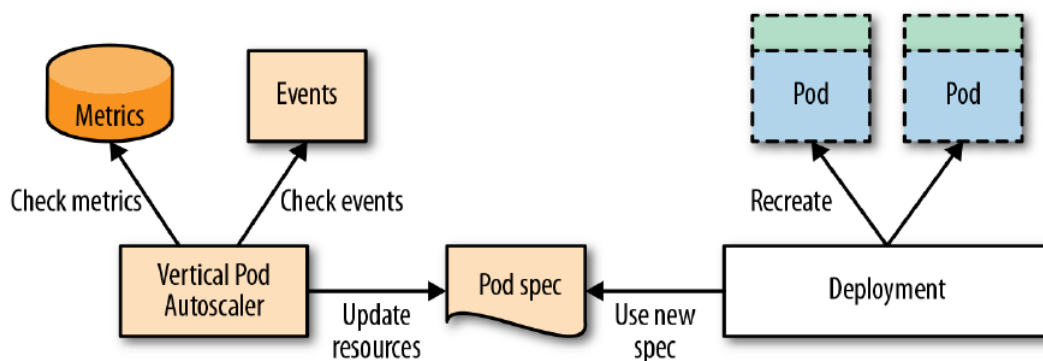
The HPA controller continuously retrieves metrics, calculates the required replicas (using a formula like  $\text{desiredReplicas} = \text{currentReplicas} \times \text{currentMetricValue} / \text{desiredMetricValue}$ ), and updates the target resource's replica count.

Autoscaling Metrics can be **Standard** (CPU, memory), **Custom** (application-specific, using Custom Metrics API, potentially from adapters like Prometheus), or **External** (resources outside the cluster, like queue depth).

Key considerations for HPA include choosing metrics directly correlated to Pod count (like QPS or CPU usage), preventing rapid fluctuations ("thrashing"), and understanding the inherent delay in reaction time due to metric collection/analysis cycles.

**Knative** Serving offers more advanced horizontal scaling, including "scale-to-zero," leveraging service meshes like Istio.

**Vertical Pod Autoscaler (VPA):** Automatically adjusts resource requests and limits for Pods. A VPA definition targets Pods using a selector (matchLabels) and has an updatePolicy to control how changes are applied.



#### Example VPA definition:

- VPA components include a recommender (suggests resource values based on usage, stored in VPA status), an admission plugin (applies recommendations to newly created Pods), and an updater (evicts running Pods so they are recreated with updated resources).
- The updateMode controls VPA's actions:
  - Off → Recommendation only.
  - Initial → Apply changes to only to new Pods.

- Auto → evicts and recreates running Pods to apply recommendations.




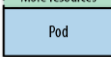


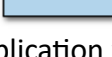
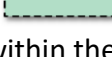
The. spec.updatePolicy.updateMode field controls how the VPA applies its recommended resource changes to Pod.

.spec.updatePolicy.updateMode Value	Description	How Changes Are Applied	Level of Potential Disruption
<b>off</b>	The VPA gathers Pod metrics and events and produces resource recommendations. The recommendations are stored in the status section of the VPA resource.	<b>Does not apply the recommendations to the Pods</b>	Nondisruptive. Useful for getting insight into Pod resource consumption without making changes.
<b>Initial</b>	Builds upon the Off mode. The VPA recommender produces recommendations. It also activates the VPA admission plugin.	<b>Applies the recommendations only to newly created Pods</b> that match the VPA's label selector. This occurs when a Pod is created, scaled manually, updated by a Deployment, or evicted and restarted. Does not restart running Pods.	<b>Partially disruptive.</b> Changes resource requests of new Pods, which can affect their scheduling and potentially lead to them being scheduled on different nodes or not scheduled at all if capacity is insufficient.
<b>Auto</b>	The <b>most disruptive</b> mode. Includes the functionality of the Initial mode <sup>6</sup> . It also activates the VPA's updater component <sup>6</sup> .	<b>Evicts running Pods</b> that match its label selector. After eviction, these Pods are recreated by the VPA admission plugin component, which updates their resource requests according to the recommendations.	Most disruptive. Restarts all Pods to forcefully apply recommendations, which can lead to unexpected scheduling issues and service disruptions.

- **VPA can be disruptive as it often requires Pod recreation.**
- **VPA and HPA can have coexistence issues if both try to manage the same resource metrics.**
- **Cluster Autoscaler (CA):** Automatically adjusts the number of nodes in the Kubernetes cluster. Works with cloud providers that support provisioning/decommissioning nodes on demand. The Cluster API project aims to standardise this across providers.
  - **Scale-up:** Triggered when Pods are unschedulable because no existing node has enough capacity. CA finds a suitable node group and requests a new node. Expanders can be configured to choose node groups based on cost, resource waste, etc..
  - **Scale-down:** Triggered when nodes are underutilized (e.g., >50% capacity unused) and all Pods on the node can be moved elsewhere. CA performs simulation to ensure Pods can be rescheduled. Various criteria prevent node deletion (e.g., Pods with local storage, Pods violating PodDisruptionBudget). The node is marked unschedulable, Pods are evicted, and the node is deleted.
  - **CA is complementary to HPA/VPA**, providing capacity elasticity at the cluster level.

#### Application Scaling Levels

- Scaling can be viewed across different levels, from granular application tuning to coarse-grained cluster scaling.

Technique	Action	Scaling Example	
Application Tuning	Tune process (threads, heap, etc.)		
Vertical Pod Autoscaler	Increase/reduce container resources		
Horizontal Pod Autoscaler	Add/remove Pods		
Cluster Autoscaler	Add/remove Nodes		

- **Application Tuning:** Optimising the application within the container (e.g., thread pools, memory sizes). This is foundational and affects the effectiveness of other scaling methods. Start scripts or in-app autoscaling libraries can help.
- **Vertical Pod Autoscaling:** *Setting container resource requests and limits based on actual usage.* VPA automates this but can be disruptive.
- **Horizontal Pod Autoscaling:** Adjusting the number of Pod replicas. Generally less disruptive than vertical scaling and the most popular method.
- **Cluster Autoscaling:** Scaling the underlying nodes. Provides capacity elasticity beyond the current cluster size.

#### Discussion Points

- Elasticity is an evolving area, with ongoing developments in HPA, VPA, and new projects like Knative introducing features like scale-to-zero.
- Configured correctly, Kubernetes can make a system antifragile, meaning it gets bigger and stronger under heavy load instead of breaking.



## Chapter 25 Image Builder

The Image Builder pattern focuses on building container images directly within a Kubernetes cluster.

Why build images within the cluster?

- Reduced maintenance costs: Having one cluster for both building and running applications can significantly cut down on maintenance expenses.
- Simplified capacity planning: Both build and runtime phases share the same infrastructure, streamlining capacity planning.
- Automated redeployment: If a base image changes (e.g., due to a security vulnerability fix), the cluster, being aware of both the build and deployment, can automatically trigger a redeployment of affected applications.
- Efficient scheduling: Kubernetes' sophisticated scheduler is well-suited for efficiently finding computing resources for build jobs, which is a common challenge in Continuous Integration (CI) systems.
- Daemonless Builds: Building in-cluster necessitates higher security. Many tools now allow daemonless builds (builds without root privileges), reducing the attack surface by not requiring the Docker daemon to run with elevated permissions. Prominent daemonless tools include img, buildah, and Kaniko.

Techniques for building images within Kubernetes:

1. OpenShift Build Subsystem: This is an older and more mature method for in-cluster image building, specific to Red Hat OpenShift, an enterprise distribution of Kubernetes.

Build Strategies supported by OpenShift:

- **Source-to-Image (S2I):** Takes application source code and, with a language-specific S2I builder image, creates a runnable artifact, then pushes it to the integrated registry.
- **Docker Builds:** Uses a Dockerfile and context directory to create an image, similar to how a Docker daemon would.
- **Pipeline Builds:** Maps a build to internally managed Jenkins server jobs, allowing for Jenkins pipeline configuration.
- **Custom Builds:** Provides full control over image creation within the build container.
- **Input Sources for builds:** Git Repository, Dockerfile, Image (for chained builds), Secret, and Binary.
- **ImageStream:** An OpenShift resource referencing container image, enabling OpenShift to emit events when an image is updated, which can trigger new builds or deployments.
- **DeploymentConfig:** OpenShift's alternative to Kubernetes Deployment, which directly integrates with ImageStreams.
- **Triggers:** Mechanisms like imageChange react to ImageStreamTag changes, leading to automatic rebuilds or redeployments.

Source-to-Image (S2I) Details: An S2I builder image is a standard container image with S2I scripts, including mandatory assemble (for building) and run (for entry point) commands.

- For example: An S2I build can use a Java builder image to compile source from GitHub and push the output to an ImageStreamTag. It can be automatically rebuilt when the builder image updates.
- Drawbacks of S2I: Can be slow for complex applications without optimisation (e.g., caching dependencies with an internal Maven repository) and the generated image contains the whole build environment, increasing size and attack surface.
- Chained Builds: Addresses S2I drawbacks by using a second build (often a Docker build) to take the artifacts from an S2I build and create a slim runtime image. This second build can be triggered automatically by changes in the S2I build's output ImageStream.

Knative Build:

- A newer project by Google (started in 2018) for bringing advanced application functionality to Kubernetes, *built on top of service meshes* like Istio.
- It provides services primarily for application developers, including Knative Serving (for scale-to-zero), Knative Eventing (for event delivery), and Knative Build (for compiling source code to images).
- Knative Build is essentially about transforming source code into a runnable container image and pushing it to a registry.
- Build CRD: The central element, defining concrete steps for the build, including source (e.g., Git) and build steps (each referring to a builder image).

*For Example:* A Knative Java build uses Maven with Jib to compile and push an image from a GitHub source, with build steps accessing a shared /workspace volume. Under the Hood: A Knative Build resource is transformed into a Kubernetes Pod, with build steps translated into a chain of Init Containers that execute sequentially. The primary container then does nothing.

- **Build Templates:** The BuildTemplate custom resource can be used to reuse common build steps for similar builds, supporting parameters for customisation.

*NOTE: Current State: Knative Build is still a young project and is expected to be replaced by Tekton Pipelines.*

The Image Builder pattern is vital for Continuous Delivery, allowing for tighter integration between development and operations by bringing the build process directly into the Kubernetes cluster.

## Future Read

- Principles of Container-Based Application Design
- The Twelve-Factor App
- Domain-Driven Design: Tackling Complexity in the Heart of Software
- Container Best Practices

- Best Practices for Writing Dockerfiles
- Container Patterns
- General Container Image Guidelines
- Knative
- Kubernetes Best Practices:
  - Resource Requests and Limits
- Kubernetes Documentation
  - Pods
  - Using ConfigMap
  - Resource Quota
  - **Pod Priority and Preemption** : The priority class to use with this Pod, as defined in PriorityClass resource
- Calculates the required number of replicas based on the current metric value and targeting the desired metric value. Here is a simplified version of the formula.

$$desiredReplicas = \left\lceil currentReplicas \times \frac{currentMetricValue}{desiredMetricValue} \right\rceil$$

- Different metrics available for autoscaling:

Metric Type	How Declared	Description/Purpose	Source/Details
<b>Standard metrics</b>	<code>.spec.metrics.resource[:].type</code> set to Resource	Represent <b>resource usage metrics</b> like <b>CPU</b> and <b>memory</b> . Generic and available across clusters. Values are based on the <b>container resource requests</b>	Can be specified as a <b>percentage</b> or <b>absolute</b> value. Typically provided by the metrics server or Heapster
<b>Custom metrics</b>	<code>.spec.metrics.resource[:].type</code> set to Object or Pod	Describe Pod-specific (Pod type) or any other object (Object type) metrics. Require an advanced monitoring setup	Served in an aggregated API Server under the <code>custom.metrics.k8s.io</code> API path. Provided by various metrics adapters (e.g., Prometheus, Datadog)
<b>External metrics</b>	Not explicitly mentioned in the sources how declared via <code>.spec.metrics</code> , but described as a category.	Describe <b>resources outside of the Kubernetes cluster</b> . Useful for scaling based on external factors like queue depth	Populated by an external metrics plugin, similar to custom metrics. Example given is scaling based on messages in a cloud-based queue

- Difference between Labels and Annotations?

Label	Annotation
The main purpose of Labels is for Identification, Selection, and grouping	The main purpose of Annotation is for metadata storage and configuration that don't affect object selection.

<p>These are <b>Queryable</b>: Can be used in selectors to find and filter resources</p> <p><b>Actionable</b>: Used by Kubernetes controllers and services to make decisions</p> <p><b>Limited</b>: Key-value pairs with restrictions on characters and length</p>	<p>There are Not queryable: Cannot be used in selectors.</p> <p>Informational: Store arbitrary metadata, configuration, or external tool data</p> <p>And</p> <p>Flexible: No restrictions on content, can store large amounts of data.</p>
<pre> metadata:   labels:     app: nginx     version: v1.2.3     environment: production     tier: frontend </pre>	<pre> metadata:   annotations:     kubernetes.io/change-cause: "Updated to version1"     deployment.kubernetes.io/revision: "3"     ingress.kubernetes.io/ssl-redirect: "true"     k8spatterns.io/podDeleteSelector: "app=webapp"     build-info: "Built by Jenkins job #123 on 2024-01-15" </pre>
<p><b>Use Cases</b></p> <p><b>Service selection</b>: Services use label selectors to find target Pods.</p> <p><b>Deployment targeting</b>: ReplicaSets use labels to manage Pods</p> <p><b>Resource queries</b>: <code>kubectl get pods -l app=nginx</code></p>	<p><b>Use Case</b></p> <p><b>Tool configuration</b>: Ingress controllers read annotations for SSL settings</p> <p><b>Metadata storage</b>: Build information, contact details, documentation links</p> <p><b>Custom controller logic</b>: Like the example you shared where annotations trigger Pod deletion</p> <p><b>Operational data</b>: Last deployment cause, revision numbers</p>