# IBM Introduction to Containers w/ Docker, Kubernetes & OpenShift

## Module 1: Containers and Containerization

### 1. Giới thiệu

#### Container và Docker:

- Container không phải là công nghệ mới, nhưng nổi bật từ 2013 với Docker.
- Khảo sát CNCF (2016–2019) cho thấy tỉ lệ sử dụng container trong sản xuất tăng từ 23% → 84%.
- Container trở thành **công nghệ gốc cloud tiêu chuẩn**, không phải xu hướng nhất thời.

#### Kubernetes:

- Nền tảng mã nguồn mở do Google phát triển và đóng góp cho cộng đồng năm 2015.
- Dùng để **sắp xếp và quản lý container**.
- Số lượng quan tâm và người dùng Kubernetes tăng mạnh (58% → 78% theo khảo sát CNCF 2019).
- Hệ sinh thái Kubernetes phát triển mạnh, bao gồm các công cụ bảo mật đám mây, mạng dịch vụ, Red Hat OpenShift, Istio.

### 2. Container

**Container** là đơn vị phần mềm tiêu chuẩn đóng gói **mã ứng dụng, runtime, thư viện hệ thống và cài đặt cần thiết**, giúp ứng dụng chạy nhất quán trên nhiều nền tảng.

#### Lợi ích container:

- Nhẹ, nhanh, cô lập, di động, an toàn.
- Chạy trên nhiều nền tảng: Windows, Linux, Mac OS, đám mây, tại chỗ.
- Hỗ trợ nhiều ngôn ngữ lập trình (Python, Java, Node...).
- Giảm thời gian và chi phí triển khai, cải thiện sử dụng CPU/bộ nhớ.
- Tự động hóa quy trình và hỗ trợ ứng dụng thế hệ mới (microservices).

#### Vấn đề trong môi trường truyền thống:

- Ứng dụng không cô lập, tài nguyên bị sử dụng kém hoặc quá tải.
- Triển khai tốn kém, bảo trì phức tạp, khả năng mở rộng hạn chế.
- Khó triển khai ứng dụng trên nhiều môi trường và hệ điều hành.

#### Thách thức khi dùng container:

- Quản lý hàng nghìn container có thể phức tạp.
- Di chuyển các ứng dụng kế thừa (monolithic) khó khăn.
- Xác định kích thước vùng chứa phù hợp đôi khi phức tạp.
- Bảo mật máy chủ cần quan tâm nếu OS bị ảnh hưởng.

#### Các nhà cung cấp container phổ biến:

- **Docker:** Nền tảng container phổ biến nhất.

- **Podman:** Không daemon, an toàn hơn Docker.
- **LXC:** Phù hợp ứng dụng và workload nhiều dữ liệu.
- **Vagrant:** Cách ly cao trên máy vật lý.

## 3. Docker

### Khái niệm Docker:

- **Docker** là nền tảng mở để **phát triển, vận chuyển và chạy ứng dụng dưới dạng container**.
- Nổi bật nhờ **kiến trúc đơn giản, khả năng mở rộng và tính di động** trên nhiều nền tảng, môi trường và vị trí.
- Cách ly ứng dụng khỏi cơ sở hạ tầng: phần cứng, hệ điều hành và runtime container.
- Viết bằng **ngôn ngữ Go** và sử dụng **tính năng kernel Linux**, đặc biệt là **namespace** để tạo không gian làm việc biệt lập cho từng container.

### Công nghệ và hệ sinh thái Docker:

- Các công cụ bổ sung: Docker CLI, Docker Compose, Prometheus.
- Các plugin: lưu trữ, networking...
- Tích hợp với các công cụ phối hợp như Docker Swarm, Kubernetes.
- Hỗ trợ các phương pháp phát triển hiện đại: **microservices, serverless**.

### Lợi ích Docker:

- Môi trường nhất quán và cô lập → triển khai ổn định.
- Hình ảnh Docker nhỏ, có thể tái sử dụng → tăng tốc phát triển.
- Tích hợp tốt với **Agile, CI/CD, DevOps**.
- Quản lý phiên bản dễ dàng → tăng tốc thử nghiệm, khôi phục và triển khai lại.
- Container độc lập nền tảng → di động cao.
- Dễ dàng mở rộng, phân đoạn và bảo trì ứng dụng.

### Thách thức / hạn chế:

- Không phù hợp với ứng dụng **nguyên khối (monolith)**.
- Không tối ưu cho các ứng dụng **yêu cầu hiệu suất cao, bảo mật nghiêm ngặt, giao diện GUI phong phú hoặc chức năng máy tính để bàn tiêu chuẩn**.

### Quy trình cơ bản:

1. **Viết Dockerfile:**

- Dockerfile là tệp văn bản mô tả các lệnh để tạo hình ảnh container.
- `FROM` xác định hình ảnh cơ sở, hình ảnh cơ sở là **một hệ điều hành hoặc môi trường đã được cấu hình sẵn**, chẳng hạn như Ubuntu, Alpine, Debian, hoặc một ngôn ngữ lập trình như Python, Node.js.
- `CMD` xác định lệnh khi container chạy trên terminal (ví dụ: in “hello world”).

2. **Tạo hình ảnh container:**
    - Lệnh: `docker build -t my-app .`
        - `t my-app`: gán tên (tag) cho hình ảnh.
        - `.`: chỉ thư mục hiện tại chứa Dockerfile.
    - Khi chạy lệnh build, Docker gửi ngữ cảnh build đến Docker Daemon và tiến hành tạo hình

3. **Kiểm tra hình ảnh:**
   - Lệnh: `docker images`
   - Hiển thị danh sách tất cả hình ảnh Docker trên máy, bao gồm:
     - Tên kho lưu trữ (repository).
     - Tên ứng dụng (image name).
     - Thẻ (tag).
     - ID hình ảnh (image ID).
     - Ngày tạo và kích thước hình ảnh.
4. **Tạo và chạy container:**
   - Lệnh: `docker run my-app`
     - Sử dụng hình ảnh đã tạo để khởi tạo container.
     - Container sẽ chạy và thực hiện lệnh mặc định ( `CMD` ) trong Dockerfile.
     - Có thể đặt tên container, gán cổng hoặc mount volume nếu cần.
5. **Quản lý container và hình ảnh:**
   - `docker ps` : hiển thị chi tiết các container đang chạy.
   - `docker stop` / `docker rm` : dừng hoặc xóa container.
   - `docker pull` : kéo hình ảnh từ registry về máy.
   - `docker push` : đẩy hình ảnh lên registry để chia sẻ hoặc lưu trữ.

### Các lệnh Docker quan trọng khác:

- `build` : tạo hình ảnh từ Dockerfile.
- `images` : liệt kê tất cả hình ảnh Docker có trên máy.
- `run` : tạo và chạy container từ hình ảnh.
- `pull` : tải hình ảnh từ registry.
- `push` : lưu hình ảnh lên registry.
- `ps` : kiểm tra các container đang chạy.

## 4. Docker Object

### Các đối tượng chính trong Docker:

1. **Dockerfile:**
   - Là tệp văn bản chứa các hướng dẫn để tạo hình ảnh Docker.
   - Có thể tạo bằng bất kỳ trình soạn thảo văn bản nào hoặc trực tiếp từ terminal.
   - Các lệnh cơ bản:
     - `FROM` : xác định hình ảnh cơ sở (ví dụ: hệ điều hành hoặc môi trường ngôn ngữ như Go, Node.js).
     - `RUN` : thực thi các lệnh trong quá trình xây dựng hình ảnh.
     - `CMD` : xác định lệnh mặc định khi container chạy (mỗi Dockerfile chỉ nên có một lệnh CMD cuối cùng).
2. **Hình ảnh Docker (Docker Image):**
   - Là mẫu chỉ đọc, được tạo từ Dockerfile.
   - Mỗi lệnh trong Dockerfile tạo một **lớp** mới trong hình ảnh.

- Khi cập nhật Dockerfile và xây dựng lại hình ảnh, Docker chỉ tạo lại các lớp thay đổi → tiết kiệm dung lượng và băng thông.
- Khi khởi tạo từ hình ảnh, sẽ tạo ra một **container đang chạy**.

3. **Container Docker:**
   - Là phiên bản có thể chạy của một hình ảnh Docker.
   - Quản lý container bằng **Docker CLI hoặc API**: tạo, bắt đầu, dừng, xóa.
   - Container có thể kết nối với nhiều mạng, gắn kết bộ nhớ, hoặc tạo hình ảnh mới dựa trên trạng thái hiện tại.
   - Docker giữ container cách ly với nhau và với máy chủ.
4. **Mạng Docker (Docker Networks):**
   - Giúp cô lập giao tiếp giữa các container.
   - Cho phép container trao đổi dữ liệu an toàn mà không ảnh hưởng đến các container khác.
5. **Khối lượng lưu trữ (Volumes) và ổ đĩa gắn kết:**
   - Dữ liệu trong container mặc định sẽ mất khi container dừng.
   - Docker sử dụng volumes hoặc bind mounts để **giữ dữ liệu ngay cả khi container ngừng chạy**.
6. **Plugin và tiện ích bổ sung:**
   - Bao gồm các plugin lưu trữ hoặc mạng, cung cấp khả năng kết nối với các nền tảng bên ngoài, mở rộng chức năng Docker.

### Đặt tên hình ảnh Docker:

- Định dạng: `<tên máy chủ>/<kho lưu trữ>:<thẻ>`
  - **Tên máy chủ**: chỉ định registry (ví dụ: `docker.io`).
  - **Kho lưu trữ (repository)**: nhóm các hình ảnh liên quan (ví dụ: `ubuntu`).
  - **Thẻ (tag)**: phiên bản hoặc biến thể của hình ảnh (ví dụ: `18.04`).
- Ví dụ: `docker.io/ubuntu:18.04`

## 5. Docker Architechure

### Kiến trúc

1. **Máy khách Docker (Docker Client):**
   - Giao tiếp với máy chủ Docker thông qua **CLI (Command Line Interface)** hoặc **REST API**.
   - Gửi các lệnh như `docker build`, `docker run` đến máy chủ để thực hiện.
   - Có thể chạy cùng hệ thống với daemon Docker hoặc kết nối từ xa.
2. **Máy chủ Docker (Docker Server / Docker Host):**
   - Chứa **daemon Docker** (`dockerd`) chịu trách nhiệm xử lý các lệnh Docker.
   - Thực hiện các tác vụ nặng: xây dựng, chạy và phân phối container.
   - Quản lý các đối tượng Docker: hình ảnh, container, mạng, không gian tên, volumes, plugin và tiện ích bổ sung.
   - Có thể giao tiếp với các daemon Docker khác để quản lý dịch vụ và container trên nhiều máy.
3. **Registry Docker (Docker Registry):**
   - Lưu trữ và phân phối hình ảnh container.

- Có thể là công khai (Docker Hub) hoặc riêng tư (do doanh nghiệp tự quản lý hoặc qua nhà cung cấp bên thứ ba như IBM Cloud).
- Cho phép các hệ thống cục bộ, đám mây và tại chỗ **kéo (pull) hình ảnh** hoặc **đẩy (push) hình ảnh** vào registry.

### Quy trình container hóa bằng Docker:

1. **Xây dựng hình ảnh:**
   - Sử dụng hình ảnh cơ sở hoặc Dockerfile để tạo hình ảnh container.
   - Lệnh `docker build` tạo hình ảnh với tên và tag xác định.
2. **Đẩy hình ảnh lên registry:**
   - Lệnh `docker push` gửi hình ảnh vào registry để lưu trữ và chia sẻ.
3. **Khởi chạy container:**
   - Máy chủ Docker kiểm tra hình ảnh cục bộ.
   - Nếu chưa có, máy khách Docker sẽ **kết nối registry và kéo hình ảnh về**.
   - Daemon Docker tạo ra một container đang chạy dựa trên hình ảnh.
   - Lệnh `docker run` thực hiện việc khởi chạy container từ hình ảnh.

# Module 2: Kubernetes

## 1. Container orchestration

Khi số lượng container tăng lên từ vài cái đến hàng trăm hoặc hàng nghìn, việc quản lý thủ công trở nên phức tạp và khó kiểm soát. **Container orchestration** ra đời để tự động hóa vòng đời container (triển khai, quản lý, mở rộng, kết nối mạng và đảm bảo tính sẵn sàng).

Nó giúp giảm độ phức tạp, tăng tốc độ, tính linh hoạt, hiệu quả tài nguyên, đồng thời tích hợp tốt với CI/CD và DevOps. Container orchestration có thể triển khai tại chỗ hoặc trên môi trường cloud (public, private, multi-cloud) và thường liên quan đến yêu cầu **SOAR** (security, orchestration, automation, response).

Các tính năng chính gồm: xác định image, triển khai và phân bổ tài nguyên, đảm bảo hiệu năng, tự động mở rộng, cân bằng tải, cập nhật/rollback, health check, và quản lý bằng file cấu hình (YAML/JSON).

**Lợi ích:**

- Năng suất cao hơn, giảm lỗi thủ công.
- Triển khai nhanh, phát hành tính năng liên tục.
- Giảm chi phí so với VM.
- Bảo mật mạnh hơn nhờ cô lập.
- Mở rộng dễ dàng, khôi phục lỗi nhanh, đảm bảo high availability.

## 2. Kubernetes

### Giới thiệu

**Định nghĩa**: Kubernetes (K8s) là nền tảng mã nguồn mở để tự động hóa triển khai, mở rộng và quản lý ứng dụng container. Phát triển bởi Google, nay do CNCF quản lý. Là tiêu chuẩn de facto trong container orchestration, dễ dàng triển khai on-premises hoặc cloud.

- **Khái niệm chính**:
  - **Pod**: đơn vị tính toán nhỏ nhất, chạy workload.
  - **Service**: phơi bày ứng dụng trên tập hợp pod (DNS/IP).

    - **Storage**: hỗ trợ lưu trữ tạm thời và lâu dài.
    - **Configuration**: cấu hình cho pod.
    - **Security**: kiểm soát truy cập pod & API.
    - **Policies**: đảm bảo pod khớp node.
    - **Scheduling & Eviction**: phân bổ tài nguyên, loại bỏ pod khi thiếu.
    - **Preemption**: ưu tiên pod quan trọng.
    - **Cluster administration**: quản lý, tạo cụm.
- **Khả năng**:
    - Triển khai & rollback tự động.
    - Health monitoring, tự phục hồi container hỏng.
    - Storage orchestration (local, network, cloud).
    - **Horizontal scaling** dựa trên metrics.
    - **Automated bin packing**: tăng hiệu quả sử dụng tài nguyên.
    - Secret & config management (password, token, SSH key).
    - Hỗ trợ IPv4/IPv6.
    - Batch & CI workloads.
    - Service discovery & load balancing.
    - Extensibility: thêm tính năng mà không cần sửa code gốc.

**Hệ sinh thái Kubernetes**:

- **Cloud providers**: Google, AWS, IBM, Prisma.
- **Frameworks**: Red Hat, VMware, Docker, Cloud Foundry...
- **Management**: Digital Ocean, Loodse, Tectonic...
- **Tools**: JFrog, Bitnami, Cloud66...
- **Monitoring/logging**: Grafana, Datadog, New Relic, Dynatrace...
- **Security**: Aqua, Twistlock, BlackDuck, Cilium...
- **Load balancing**: NGINX, VMware, AVI Networks.

## Kiến trúc

Một quá trình triển khai Kubernetes còn được gọi là một Kubernetes Cluster. Cluster bao gồm **Control Plane (Master Node)** và **Worker Nodes**.

- **Control Plane**: ra quyết định và duy trì trạng thái mong muốn của cluster (giống như thermostat).
- **Worker Nodes**: nơi chạy ứng dụng của người dùng (Pods/Containers).

**Thành phần Control Plane**

1. **API Server (kube-apiserver)**
    - Cổng giao tiếp chính, xử lý mọi lệnh và yêu cầu qua Kubernetes API.
    - Có thể scale ngang bằng cách chạy nhiều instance.
2. **etcd**
    - Kho dữ liệu phân tán key-value, lưu trữ toàn bộ trạng thái cluster.
    - Ghi nhận cấu hình deployment và định nghĩa trạng thái mong muốn.

3. **Scheduler (kube-scheduler)**
    - Gán pod mới tạo vào node phù hợp nhất dựa trên tài nguyên, cấu hình và nguyên tắc scheduling.
4. **Controller Manager**
    - Chạy các controller để giám sát và đảm bảo trạng thái thực tế khớp trạng thái mong muốn.
5. **Cloud Controller Manager**
    - Liên kết Kubernetes với API của nhà cung cấp cloud.
    - Cho phép Kubernetes hoạt động **cloud-agnostic** (không phụ thuộc nhà cung cấp).

**Thành phần Worker Plane**

1. **Node**
    - Máy vật lý hoặc ảo, chứa tài nguyên để chạy ứng dụng.
    - Gồm nhiều **Pods** (mỗi pod có 1 hoặc nhiều container).
2. **Kubelet**
    - Thành phần quan trọng nhất trên node.
    - Nhận cấu hình pod từ API Server, khởi chạy container thông qua container runtime, giám sát và báo cáo trạng thái pod.
3. **Container Runtime**
    - Chịu trách nhiệm tải image và chạy container.
    - Hỗ trợ nhiều runtime qua **Container Runtime Interface (CRI)**: Docker, Podman, CRI-O...
4. **Kube-proxy**
    - Proxy mạng chạy trên mỗi node.
    - Thiết lập network rules cho phép giao tiếp giữa pods và giữa workload trong cluster.

# 3. Kubernetes Objects

## Khái niệm

- Kubernetes objects = **thực thể bền vững (persistent entities)** trong cluster.
- Mỗi object có:
    - **Spec** (do người dùng cung cấp) → trạng thái mong muốn.
    - **Status** (do Kubernetes cung cấp) → trạng thái hiện tại.
- Kubernetes sẽ làm việc để đưa trạng thái thực tế về trạng thái mong muốn.

## Các thành phần cơ bản

1. **Labels & Label Selectors**
    - Labels = cặp key–value gắn vào object để phân loại.
    - Nhiều object có thể chia sẻ label.
    - Label selectors = cách chính để nhóm và chọn object.
2. **Namespaces**
    - Cơ chế cô lập nhóm tài nguyên trong một cluster.
    - Hữu ích khi nhiều nhóm/người dùng chia sẻ cluster.
    - Ví dụ:
        - `kube-system`: cho các thành phần hệ thống.

    - `default`: chứa ứng dụng của người dùng.
  - Mỗi namespace đảm bảo tên object là duy nhất trong phạm vi của nó.
3. **Pods**
  - Đơn vị triển khai nhỏ nhất, đại diện cho **một tiến trình/instance ứng dụng**.
  - Một pod bao gồm 1 hoặc nhiều container.
  - Replica pods = mở rộng ứng dụng theo chiều ngang.
  - Được định nghĩa bằng file **YAML** (có trường `kind`, `spec`, `containers`, `image`, `ports`).
4. **ReplicaSet**
  - Đảm bảo có số lượng pod bản sao (replicas) nhất định luôn chạy.
  - Cấu hình gồm `replicas`, `selector` và `pod template`.
  - Tự động tạo hoặc xóa pod để khớp số replicas yêu cầu.
  - Thường **không tạo trực tiếp** → được quản lý bởi **Deployment**.
5. **Deployment**
  - **Cấp cao hơn ReplicaSet**, quản lý pods và replicas.
  - Hỗ trợ:
    - Tự động mở rộng,
    - Cập nhật pods,
    - **Rolling updates** (triển khai phiên bản mới dần dần, hạ phiên bản cũ).
  - Phù hợp cho **ứng dụng stateless** (ứng dụng stateful dùng StatefulSet).

## Service

- Là **REST object**, trừu tượng hóa tập hợp Pods.
- Cung cấp chính sách truy cập, cân bằng tải giữa Pods.
- Có **IP duy nhất** và có thể gắn **DNS name** → không cần service discovery riêng.
- Hỗ trợ nhiều giao thức (TCP mặc định, UDP...).
- Giải quyết vấn đề **Pods thường xuyên thay đổi IP**.

**Các loại Service**

1. **ClusterIP (mặc định)**
  - Chỉ truy cập **nội bộ trong cluster**.
  - Dùng cho giao tiếp giữa các thành phần (frontend ↔ backend).
2. **NodePort**
  - Mở dịch vụ qua một **port tĩnh trên mỗi node**.
  - Tự động chuyển tiếp đến ClusterIP service.
  - Dùng trong dev/test, **không khuyến nghị production** (thiếu bảo mật & load balancing).
3. **LoadBalancer (ELB)**
  - Tích hợp NodePort + ClusterIP.
  - Tạo load balancer bên ngoài (thường từ cloud provider) để truy cập Internet.
4. **ExternalName**
  - Không dùng selector, ánh xạ đến **DNS name** ngoài cluster.

- Trả về bản ghi **CNAME**.
- Dùng để kết nối dịch vụ ngoài (ví dụ: external storage, cross-namespace).

**Ingress**

- Là API object + controller, cung cấp **quy tắc định tuyến** để truy cập nhiều dịch vụ từ ngoài cluster.
- Expose qua **HTTP (80)** và **HTTPS (443)**.
- So với ELB: tiết kiệm chi phí, quản lý trong cluster.

**Các đối tượng nâng cao**

1. **DaemonSet**
   - Đảm bảo mỗi node chạy ít nhất một Pod.
   - Dùng cho **log collection, monitoring, storage agent**.
   - Khi node mới thêm vào → pod tự động tạo trên node đó.
2. **StatefulSet**
   - Dành cho **ứng dụng stateful** (cần dữ liệu ổn định).
   - Cung cấp **danh tính cố định (sticky identity)** cho mỗi Pod.
   - Hỗ trợ **persistent storage** và đảm bảo **thứ tự triển khai**.
3. **Job**
   - Tạo Pod và theo dõi tiến trình cho đến khi hoàn tất.
   - Pod sẽ được **retry đến khi thành công**.
   - Có thể chạy nhiều Pod song song.
   - **CronJob** = Job chạy theo lịch lặp lại.

# 4. Kubectl – Kubernetes CLI

## Định nghĩa

- Kubectl = **Kubernetes Command Line Interface** (CLI).
- Dùng để **triển khai ứng dụng, quản lý tài nguyên, xem logs, scale pods, xóa resources**, v.v.
- Cấu trúc lệnh:

  ```
  kubectl <command> <type> <name> [flags]
  ```

  - **command**: thao tác (create, get, apply, delete...).
  - **type**: loại resource (pod, deployment, service...).
  - **name**: tên resource (có thể bỏ qua).
  - **flags**: tham số tùy chọn.

## 3 loại lệnh Kubectl

1. **Imperative Commands**
   - Ví dụ:

     ```
     kubectl run mypod --image=nginx
     ```

   - Ưu điểm: dễ học, nhanh, tiện cho dev/test.

- Nhược điểm:
  - Không có audit trail.
  - Không lưu cấu hình (người khác phải nhớ lại lệnh gốc).
  - Không phù hợp production.

2. **Imperative Object Configuration**
   - Dùng file **YAML/JSON** mô tả resource.
   - Ví dụ:

     ```
     kubectl create -f nginx.yaml
     ```

   - Ưu điểm:
     - Có template, dễ tái sử dụng nhiều môi trường.
     - Có thể lưu trong Git (audit trail, review).
   - Nhược điểm: cần biết schema object & viết YAML/JSON đầy đủ.
3. **Declarative Object Configuration** (khuyên dùng trong production)
   - Định nghĩa **trạng thái mong muốn** trong file cấu hình.
   - Kubectl tự xác định thao tác cần làm để đồng bộ trạng thái.
   - Ví dụ:

     ```
     kubectl apply -f configs/
     ```

   - Ưu điểm:
     - Tự động hóa, chỉ cần file config là "nguồn sự thật duy nhất" (single source of truth).
     - Thích hợp CI/CD & production.

### Một số lệnh phổ biến

- **get**: xem resource

  ```
  kubectl get pods --all-namespaces
  kubectl get services
  ```

- **apply**: tạo/cập nhật từ file

  ```
  kubectl apply -f nginx-deployment.yaml
  ```

- **delete**: xóa resource

  ```
  kubectl delete pod mypod
  ```

- **scale**: thay đổi số replicas

  ```
  kubectl scale deployment my-dep --replicas=3
  ```

- **autoscale**: bật auto-scaling cho deployment/replicaset.

# Module 3: Managing Applications with Kubernetes

# 1. Replica Set

### Vấn đề của Pod đơn lẻ:

- Không đáp ứng được khi tải tăng.
- Có nguy cơ mất điện hoặc sự cố dẫn đến downtime.
- Không đảm bảo khả năng chịu lỗi (single point of failure).

### Cách ReplicaSet hoạt động

- ReplicaSet liên tục giám sát trạng thái của các Pod và so sánh với cấu hình trong trường **spec.replicas** (số lượng bản sao mong muốn).
- Nếu số Pod thực tế ít hơn số Pod mong muốn, ReplicaSet sẽ tạo thêm Pod mới.
- Nếu số Pod nhiều hơn mức cấu hình, ReplicaSet sẽ tự động xóa bớt để đưa trạng thái thực tế về đúng với trạng thái mong muốn.
- Khi một Pod bị lỗi hoặc bị xóa thủ công, ReplicaSet sẽ ngay lập tức thay thế bằng một Pod mới để không làm gián đoạn dịch vụ.

ReplicaSet không sở hữu Pod trực tiếp. Thay vào đó, nó dựa vào **labels và selectors** để xác định tập hợp Pod mà nó cần quản lý. Điều này cho phép ReplicaSet linh hoạt trong việc chọn Pod và duy trì số lượng mong muốn.

### Mối quan hệ giữa ReplicaSet và Deployment

Mặc dù có thể tạo ReplicaSet trực tiếp từ một file YAML hoặc lệnh CLI, thực tế **ReplicaSet thường được tạo ra và quản lý thông qua Deployment**. Lý do là vì Deployment cung cấp nhiều tính năng nâng cao hơn, ví dụ:

- **Rolling updates**: cập nhật ứng dụng mà không gây downtime.
- **Rollback**: quay lại phiên bản cũ khi có sự cố.
- **Quản lý phiên bản**: lưu lại lịch sử cập nhật để dễ dàng kiểm soát.

Do đó, ReplicaSet được coi là thành phần cốt lõi nằm bên dưới Deployment, còn Deployment là công cụ bậc cao hơn giúp người dùng quản lý dễ dàng và hiệu quả hơn.

### Thao tác với ReplicaSet

1. **Tạo ReplicaSet**
   - Bằng YAML:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-rs
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
```

```
      - name: nginx
        image: nginx:latest
```

Lệnh áp dụng:

```
kubectl apply -f nginx-rs.yaml
```

2. **Kiểm tra ReplicaSet và Pod**

   ```
   kubectl get rs
   kubectl get pods
   ```

   Kết quả hiển thị số lượng Pod đang chạy và chi tiết về ReplicaSet.

3. **Mở rộng quy mô**

   Sử dụng lệnh scale:

   ```
   kubectl scale deployment hello-kubernetes --replicas=3
   ```

   ReplicaSet sẽ tự động tạo thêm Pod để đạt đủ số lượng.

4. **Xử lý Pod bị xóa**

   Nếu bạn xóa một Pod thủ công bằng lệnh:

   ```
   kubectl delete pod <pod-name>
   ```

   ReplicaSet sẽ ngay lập tức tạo ra một Pod mới thay thế để giữ nguyên trạng thái mong muốn.

5. **Xử lý Pod dư thừa**

   Nếu người dùng tự ý tạo thêm Pod vượt ngoài số lượng mong muốn, ReplicaSet sẽ xóa Pod đó để đảm bảo cụm vẫn duy trì đúng cấu hình ban đầu.

### Lợi ích của ReplicaSet

- **Đảm bảo tính khả dụng cao**: Luôn có Pod thay thế khi Pod gặp sự cố.
- **Tự động cân bằng tải**: Khi có nhiều Pod, lưu lượng sẽ được phân phối đồng đều.
- **Hỗ trợ mở rộng linh hoạt**: Có thể tăng/giảm số Pod chỉ với một lệnh.
- **Duy trì trạng thái mong muốn**: Luôn đồng bộ giữa cấu hình khai báo và trạng thái thực tế.
- **Nền tảng cho các đối tượng cao hơn**: ReplicaSet là cơ sở để Deployment hoạt động hiệu quả.

## 2. Autoscaling

Tự động mở rộng quy mô (autoscaling) là quá trình hệ thống **tự điều chỉnh tài nguyên tính toán** như Pod hoặc node sao cho phù hợp với khối lượng công việc. Mục tiêu chính:

- Đảm bảo ứng dụng luôn có đủ tài nguyên để hoạt động mượt mà.
- Tránh lãng phí tài nguyên, từ đó giảm chi phí vận hành.
- Tăng tính sẵn sàng và khả năng phản ứng nhanh với thay đổi tải.

Trong Kubernetes, autoscaling có thể xảy ra ở **hai lớp**:

1. **Cấp Pod**: thay đổi số lượng Pod hoặc tài nguyên trong Pod.
2. **Cấp Cụm (Cluster)**: thay đổi số lượng node trong cụm.

### Ba loại autoscaler trong Kubernetes

### 1. **Horizontal Pod Autoscaler (HPA) – Tự động mở rộng Pod theo chiều ngang**

- **Nguyên lý**: HPA điều chỉnh số lượng Pod của một workload (ví dụ: Deployment, ReplicaSet) bằng cách **tăng hoặc giảm số Pod** dựa trên số liệu giám sát (thường là CPU, bộ nhớ, hoặc custom metrics).
- **Ví dụ hoạt động**:
  - Buổi sáng: tải nhẹ → chỉ cần 1 Pod.
  - 11h trưa: lượng người dùng tăng cao → HPA tự động tăng lên 3 Pod.
  - Chiều tối: tải giảm → HPA giảm dần xuống còn 1 Pod để tiết kiệm tài nguyên.
- **Cách triển khai**: có thể dùng lệnh `kubectl autoscale` hoặc định nghĩa YAML cho HPA.
- **Ưu điểm**: linh hoạt, phù hợp với ứng dụng có thể mở rộng theo nhiều bản sao.
- **Nhược điểm**: không hữu ích với ứng dụng "stateful" hoặc yêu cầu dữ liệu không thể nhân bản dễ dàng.

### 2. **Vertical Pod Autoscaler (VPA) – Tự động mở rộng Pod theo chiều dọc**

- **Nguyên lý**: VPA không thay đổi số lượng Pod mà thay đổi **tài nguyên bên trong Pod** như CPU hoặc RAM. Nó tự động tăng hoặc giảm giới hạn/tài nguyên yêu cầu cho container.
- **Ví dụ hoạt động**:
  - Buổi sáng: tải nhẹ → Pod chỉ dùng ít CPU, RAM.
  - 11h trưa: tải cao → VPA tăng giới hạn CPU và RAM cho Pod để xử lý nhiều hơn.
  - Chiều tối: tải giảm → Pod tự động "thu nhỏ" tài nguyên.
- **Ưu điểm**: phù hợp với dịch vụ khó nhân bản, ví dụ cơ sở dữ liệu, ứng dụng xử lý dữ liệu nặng.
- **Nhược điểm**: có thể gây restart Pod khi thay đổi resource, không nên kết hợp trực tiếp với HPA trên cùng một metric (CPU/memory).

### 3. **Cluster Autoscaler (CA) – Tự động mở rộng ở cấp cụm**

- **Nguyên lý**: CA điều chỉnh **số lượng node trong cụm**. Khi các Pod không thể được lên lịch do thiếu tài nguyên, CA thêm node mới. Khi nhu cầu giảm, node thừa sẽ bị xóa để tiết kiệm chi phí.
- **Ví dụ hoạt động**:
  - Buổi sáng: tải nhẹ → các node hiện tại đủ xử lý.
  - 11h trưa: tải tăng, có nhiều Pod pending → CA thêm 1 node để chứa Pod mới.
  - Buổi tối: tải giảm → Pod trên node phụ bị xóa, node trống cũng bị loại bỏ.
- **Ưu điểm**: đảm bảo cụm luôn có đủ sức mạnh tính toán.
- **Nhược điểm**: thời gian thêm/xóa node lâu hơn so với HPA và VPA, phụ thuộc hạ tầng cloud.

### Kết hợp các autoscaler

Trong thực tế, ba loại autoscaler thường được dùng **kết hợp**:

- **HPA** xử lý thay đổi nhanh chóng về số lượng Pod.
- **VPA** tinh chỉnh tài nguyên bên trong Pod để phù hợp hơn.
- **CA** bổ sung hoặc giảm bớt node trong cụm khi workload vượt quá khả năng hạ tầng hiện tại.

Ví dụ: vào giờ cao điểm, HPA có thể nhanh chóng tăng số Pod. Khi toàn bộ node đầy, CA sẽ thêm node mới để Pod được phân phối. Đồng thời, VPA đảm bảo từng Pod sử dụng tài nguyên một cách hợp lý.

## 3. Deployment strategies

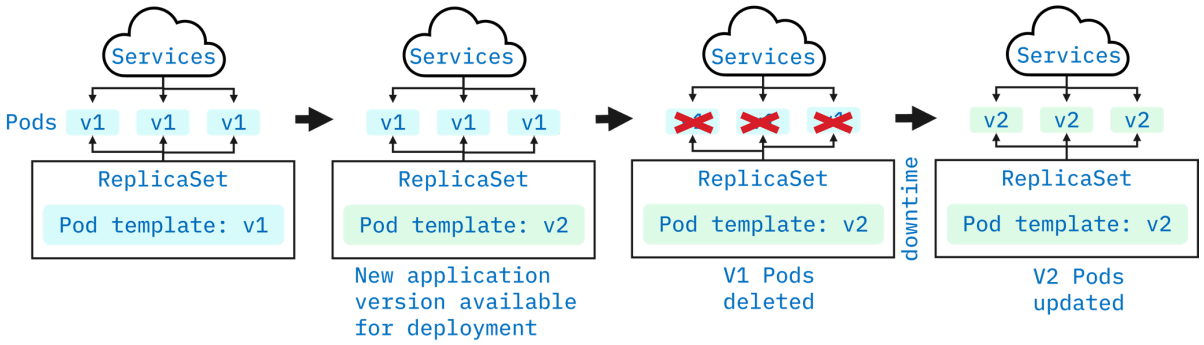
Deployment strategy trong Kubernetes là cách tiếp cận để triển khai, cập nhật hoặc rollback ứng dụng nhằm đảm bảo hệ thống luôn ở trạng thái mong muốn một cách **tự động và an toàn**. Chiến lược triển khai hiệu quả

giúp **giảm thiểu rủi ro, hạn chế downtime, tối ưu chi phí và nâng cao trải nghiệm người dùng**.

Các mục tiêu chính:

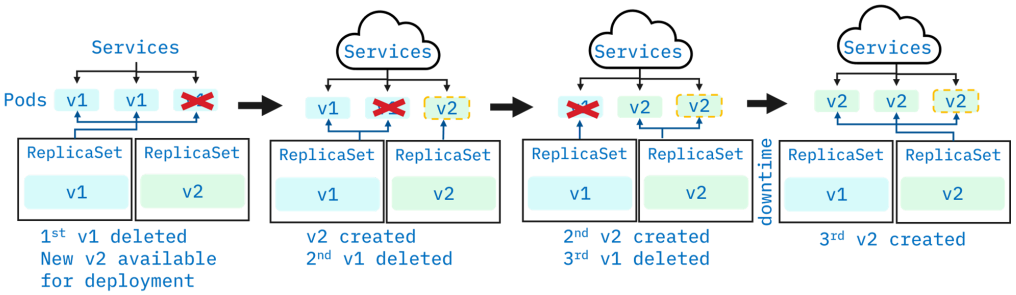
- Triển khai, cập nhật, rollback **ReplicaSets, Pods, Services, Applications**.
- **Pause/Resume** (tạm dừng/tiếp tục) triển khai.
- Thực hiện **scaling thủ công hoặc tự động**.
- Cho phép kiểm thử, giám sát hiệu năng và phản ứng nhanh với sự cố.

## Recreate



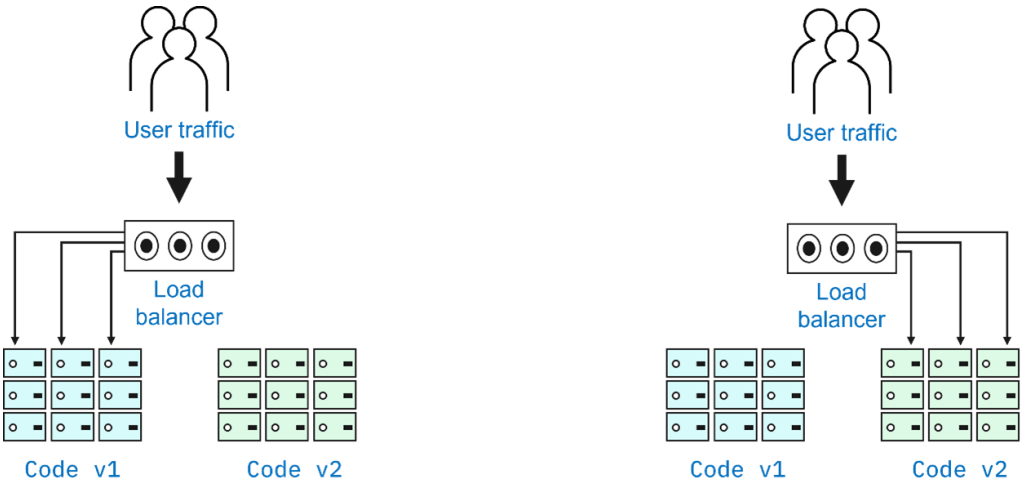

- **Cách hoạt động**: Xóa toàn bộ Pod cũ (v1), sau đó tạo Pod mới (v2).
- **Ưu điểm**: Cấu hình đơn giản, toàn bộ ứng dụng được thay thế hoàn toàn.
- **Nhược điểm**: Có downtime ngắn, không phù hợp với ứng dụng quan trọng.
- **Thích hợp khi**: Ứng dụng không quan trọng, downtime có thể chấp nhận được.

## Rolling (Ramped)



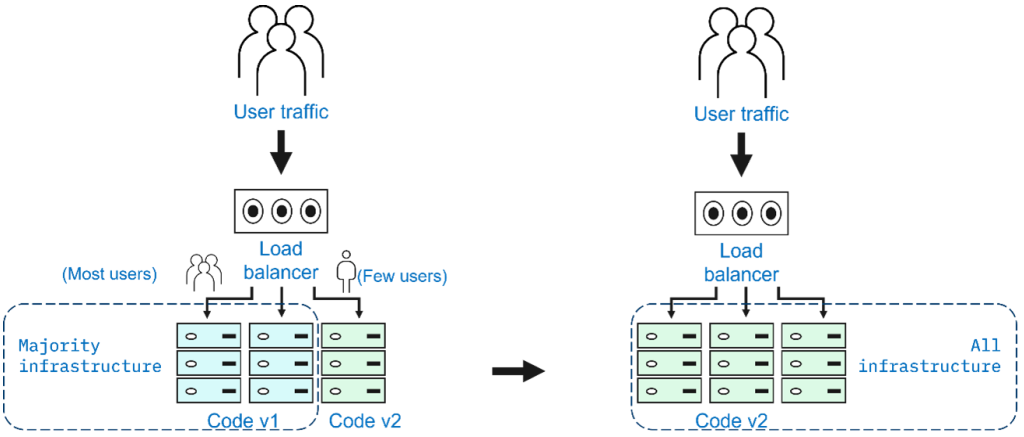

- **Cách hoạt động**: Thay thế dần từng Pod v1 bằng Pod v2 cho đến khi toàn bộ cập nhật xong.
- **Ưu điểm**: Gần như không downtime, rollback dễ dàng.
- **Nhược điểm**: Rollout/rollback mất thời gian, khó kiểm soát phân phối traffic.
- **Thích hợp khi**: Ứng dụng stateful cần đảm bảo liên tục, nhưng chấp nhận rollout chậm.

## Blue/Green



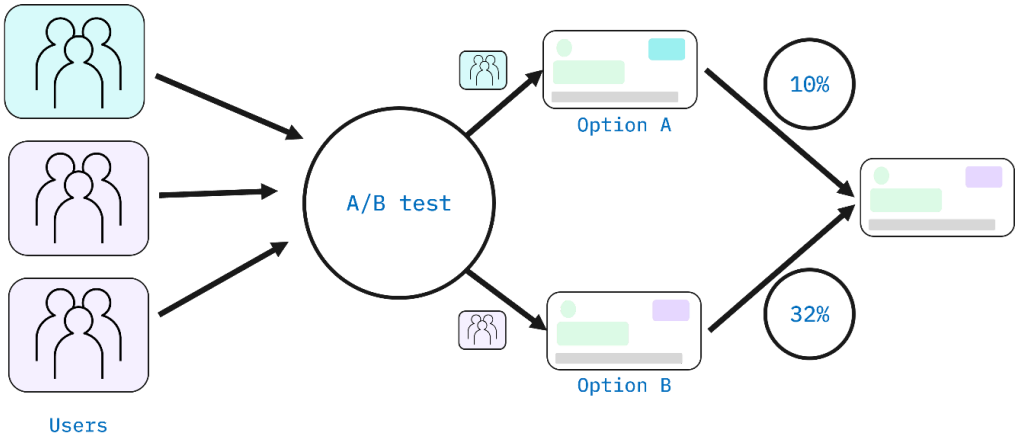

- **Cách hoạt động**: Tạo môi trường **Green** (chứa phiên bản mới) song song với môi trường **Blue** (đang chạy). Khi Green sẵn sàng, chuyển toàn bộ traffic từ Blue → Green.
- **Ưu điểm**: Rollout/rollback tức thì, không downtime.
- **Nhược điểm**: Tốn gấp đôi tài nguyên, khó xử lý ứng dụng stateful.
- **Thích hợp khi**: Ứng dụng quan trọng, cần triển khai nhanh chóng, an toàn.

## Canary



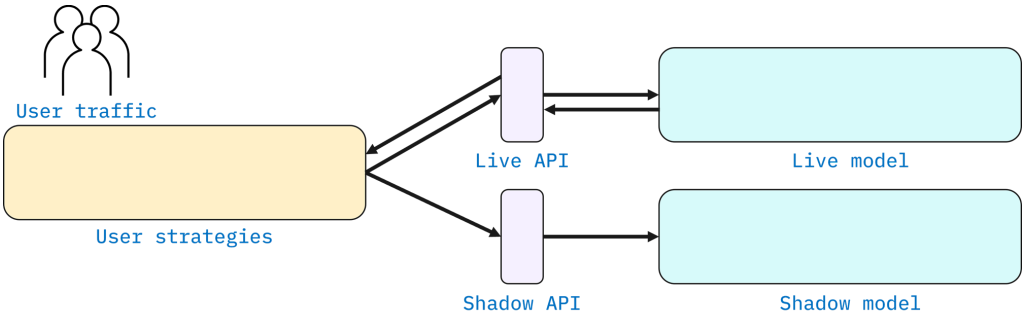

- **Cách hoạt động**: Phát hành phiên bản mới cho một **nhóm nhỏ người dùng ngẫu nhiên**, giám sát hiệu năng và lỗi. Khi ổn định, mở rộng cho toàn bộ người dùng.
- **Ưu điểm**: Rollback nhanh, dễ theo dõi hiệu năng.
- **Nhược điểm**: Rollout chậm, yêu cầu quản lý traffic hợp lý.
- **Thích hợp khi**: Muốn **zero downtime**, chấp nhận cho một số người dùng trải nghiệm sớm.

## A/B Testing




- **Cách hoạt động**: Chạy song song phiên bản A và B, phân phối cho các nhóm người dùng khác nhau theo **điều kiện cụ thể** (địa lý, cookie, OS, thiết bị...).
- **Ưu điểm**: Toàn quyền kiểm soát traffic, so sánh hành vi người dùng chi tiết.
- **Nhược điểm**: Cần load balancer thông minh, khó phân tích log/phát hiện lỗi.
- **Thích hợp khi**: Kiểm thử UI, tính năng mới dựa trên hành vi người dùng.

### Shadow




- **Cách hoạt động**: Gửi traffic thực tế đến cả phiên bản live và phiên bản shadow, nhưng chỉ phiên bản live trả kết quả cho người dùng.
- **Ưu điểm**: Test hiệu năng với dữ liệu thật, không ảnh hưởng đến người dùng.
- **Nhược điểm**: Tốn tài nguyên gấp đôi, kết quả có thể gây hiểu nhầm (không phản hồi cho user).
- **Thích hợp khi**: Kiểm thử **ứng dụng phức tạp** trong môi trường production.

### So sánh nhanh

| Strategy | Zero Downtime | Real Traffic Test | Targeted Users | Chi phí Cloud | Rollback | Độ phức tạp |
|---|---|---|---|---|---|---|
| Recreate | × | × | × | Thấp | Nhanh | Thấp |
| Rolling | ✓ | × | × | TB | Chậm | TB |
| Blue/Green | ✓ | × | × | Cao | Nhanh | TB |
| Canary | ✓ | ✓ | × | TB | Nhanh | TB |
| A/B Testing | ✓ | ✓ | ✓ | TB | TB | Cao |
| Shadow | ✓ | ✓ | × | Cao | TB | Cao |

## 4. Rolling Updates

Rolling Update (cập nhật liên tục) là một chiến lược trong Kubernetes cho phép bạn triển khai các thay đổi ứng dụng một cách **tự động, có kiểm soát và không gây gián đoạn dịch vụ**. Thay vì dừng toàn bộ ứng dụng để thay thế bằng phiên bản mới, Kubernetes lần lượt thay thế từng pod cũ bằng pod mới cho đến khi toàn bộ ứng dụng được cập nhật. Điều này đảm bảo rằng người dùng vẫn có thể truy cập ứng dụng trong suốt quá trình triển khai.

## Cách hoạt động của Rolling Update

- **Thay thế từng pod theo thứ tự**: Kubernetes tạo một pod mới từ phiên bản ứng dụng cập nhật, sau đó loại bỏ một pod cũ. Quá trình này lặp lại cho đến khi tất cả pod được thay thế.
- **Không downtime (nếu cấu hình đúng)**: Ứng dụng vẫn duy trì số lượng pod sẵn sàng phục vụ người dùng.
- **Rollback dễ dàng**: Nếu phiên bản mới gặp lỗi, bạn có thể hoàn tác (rollback) về phiên bản trước.

## Chuẩn bị trước khi áp dụng Rolling Update

Để Rolling Update diễn ra suôn sẻ, cần chuẩn bị:

1. **Readiness Probe và Liveness Probe**
   - `readinessProbe`: đảm bảo pod chỉ được đưa vào vòng load balancing khi đã sẵn sàng.
   - `livenessProbe`: kiểm tra tình trạng pod để khởi động lại nếu cần.
2. **Cấu hình chiến lược cập nhật trong Deployment YAML**
   - **maxUnavailable**: số pod tối đa có thể không khả dụng trong quá trình cập nhật.
   - **maxSurge**: số pod bổ sung có thể tạo vượt quá số lượng mong muốn.
   - Ví dụ: `maxUnavailable: 0` và `maxSurge: 2` đảm bảo không downtime và cho phép tối đa 2 pod mới chạy song song với pod cũ.
3. **Chỉ định minReadySeconds**
   - Đảm bảo pod mới phải chạy ổn định một khoảng thời gian trước khi Kubernetes tiếp tục thay thế pod khác.

## Ví dụ triển khai Rolling Update

Giả sử bạn có một Deployment với 3 pod đang chạy ứng dụng hiển thị thông báo **“Hello World!”**. Bạn muốn cập nhật ứng dụng thành **“Hello World v2”** mà không gây gián đoạn:

- Xây dựng và tải lên Docker Hub image mới:

```
docker build -t hello-kubernetes:2.0 .
docker push hello-kubernetes:2.0
```

- Cập nhật Deployment bằng cách thay đổi image trong YAML hoặc dùng lệnh:

```
kubectl set image deployment/hello-kubernetes \
hello-kubernetes=hello-kubernetes:2.0
```

- Kubernetes sẽ lần lượt tạo pod mới (v2), loại bỏ pod cũ (v1), cho đến khi tất cả pod đều chạy phiên bản mới.

## Rollback (khôi phục phiên bản cũ)

Nếu phiên bản mới gặp lỗi:

- Sử dụng lệnh rollback:

```
kubectl rollout undo deployment/hello-kubernetes
```

- Kubernetes sẽ thay thế pod v2 bằng pod v1, đưa ứng dụng trở về trạng thái trước đó.

### Chiến lược triển khai và khôi phục

Kubernetes hỗ trợ hai cách chính:

1. **All-at-Once (tất cả cùng một lúc)**
   - Toàn bộ pod cũ bị xóa, sau đó toàn bộ pod mới được tạo.
   - Có downtime vì trong quá trình chuyển đổi sẽ không có pod sẵn sàng phục vụ.
2. **Rolling (từng bước một)**
   - Pod mới và pod cũ tồn tại song song, thay thế dần dần.
   - Không downtime, phù hợp cho ứng dụng production.

Cả triển khai và rollback đều có thể thực hiện theo hai chiến lược trên. Trong thực tế, **rolling update từng bước một** là lựa chọn phổ biến nhất để đảm bảo tính ổn định và trải nghiệm người dùng.

## 5. ConfigMaps and Secrets

Trong phát triển phần mềm hiện đại, một nguyên tắc quan trọng là **không mã hóa cứng (hardcode)** các biến cấu hình vào trong mã nguồn. Điều này giúp ứng dụng linh hoạt hơn, dễ bảo trì và có thể thay đổi cấu hình mà không cần phải biên dịch hoặc triển khai lại mã. Kubernetes hỗ trợ điều này thông qua hai đối tượng API quan trọng: **ConfigMap** và **Secret**.

### ConfigMap là gì?

- **Định nghĩa**: ConfigMap là một đối tượng Kubernetes dùng để lưu trữ dữ liệu **không nhạy cảm** dưới dạng cặp **key–value**.
- **Đặc điểm**:
  - Dữ liệu không được mã hóa hoặc bảo mật, vì vậy **không phù hợp** để lưu trữ thông tin nhạy cảm (như mật khẩu, token, API key).
  - Giúp tách biệt **cấu hình ứng dụng** khỏi **mã ứng dụng**, giúp thay đổi linh hoạt hơn.
  - Kích thước tối đa cho mỗi ConfigMap: **1 MB**. Với dữ liệu lớn hơn, bạn nên dùng **Persistent Volume**, cơ sở dữ liệu, hoặc dịch vụ lưu trữ bên ngoài.
  - Có thể chứa dữ liệu **tùy chọn**, **nhị phân**, và được chia sẻ cho nhiều pod hoặc deployment.

### Ba cách để tạo ConfigMap

1. **Tạo từ chuỗi key–value trực tiếp**

   ```
   kubectl create configmap myconfig --from-literal=message="Hello from ConfigMap"
   ```

2. **Tạo từ file hoặc thư mục**
   - Từ một file `.properties` hoặc `.env`:

     ```
     kubectl create configmap myconfig --from-file=my.properties
     ```

   - Từ một thư mục (tất cả file trong thư mục sẽ trở thành key):

     ```
     kubectl create configmap myconfig --from-file=./configdir
     ```

3. **Tạo từ YAML manifest**

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: myconfig
data:
  message: "Hello from YAML"
```

```
kubectl apply -f myconfig.yaml
```

## Cách sử dụng ConfigMap

ConfigMap có thể được gắn vào pod/deployment theo hai cách chính:

- **Dùng như biến môi trường (Environment Variable)**

```
env:
  - name: APP_MESSAGE
    valueFrom:
      configMapKeyRef:
        name: myconfig
        key: message
```

→ Trong code Node.js, bạn có thể truy cập bằng `process.env.APP_MESSAGE`.

- **Dùng như Volume Mount (gắn file)**

```
volumes:
  - name: config-volume
    configMap:
      name: myconfig
volumeMounts:
  - name: config-volume
    mountPath: /etc/config
```

→ Các key trong ConfigMap sẽ xuất hiện dưới dạng file trong thư mục `/etc/config`.

## Secret là gì?

- **Định nghĩa**: Secret là một đối tượng Kubernetes để lưu trữ dữ liệu **nhạy cảm** (mật khẩu, chứng chỉ, token API).
- **Khác với ConfigMap**:
  - Dữ liệu trong Secret được lưu trữ **dưới dạng Base64** (không phải mã hóa mạnh mẽ, nhưng ẩn thông tin khi xem YAML).
  - Hạn chế quyền truy cập tốt hơn, dùng cho các thông tin quan trọng.

## Ba cách để tạo Secret

1. **Tạo từ chuỗi key–value**

```
kubectl create secret generic api-creds --from-literal=API_KEY=abcd1234
```

2. **Tạo từ file**

```
kubectl create secret generic api-creds --from-file=./secrets.properties
```

3. **Tạo từ YAML manifest**

```
apiVersion: v1
kind: Secret
metadata:
  name: api-creds
type: Opaque
data:
  API_KEY: YWJjZDEyMzQ=   # "abcd1234" được mã hóa Base64
```

### Cách sử dụng Secret

- **Dùng như biến môi trường**

```
env:
  - name: API_KEY
    valueFrom:
      secretKeyRef:
        name: api-creds
        key: API_KEY
```

→ Trong ứng dụng: `process.env.API_KEY`.

- **Dùng như Volume Mount**

```
volumes:
  - name: secret-volume
    secret:
      secretName: api-creds
volumeMounts:
  - name: secret-volume
    mountPath: /etc/secret
    readOnly: true
```

→ Secret sẽ xuất hiện dưới dạng file, ví dụ `/etc/secret/API_KEY`.

## 6. Service Binding

### Khái niệm về Service Binding

- **Service Binding** là quá trình kết nối ứng dụng hoặc một cụm Kubernetes với các dịch vụ bên ngoài (external services) hoặc dịch vụ phụ trợ (supporting services).
- Các dịch vụ này có thể là:
  - **REST API**
  - **Cơ sở dữ liệu (database)**
  - **Event bus**
  - Hoặc dịch vụ AI/ML trên đám mây (ví dụ: IBM Watson).
- Mục tiêu:
  - Quản lý **cấu hình** và **thông tin đăng nhập** (credentials) cần thiết để ứng dụng có thể sử dụng dịch vụ.

- Đảm bảo **an toàn bảo mật**: dữ liệu nhạy cảm (API key, password, token...) được lưu trong **Kubernetes Secret** thay vì hard-code vào mã nguồn.

## Cách hoạt động của Service Binding

Quy trình cơ bản:

1. **Tạo phiên bản dịch vụ**: Cung cấp một instance của dịch vụ bên ngoài (ví dụ: IBM Watson Tone Analyzer).
2. **Liên kết dịch vụ với cụm Kubernetes**: Dùng lệnh `service bind` để gắn dịch vụ đó với cluster.
   - Khi liên kết xong, hệ thống sẽ tự động sinh ra một **Kubernetes Secret** chứa thông tin đăng nhập.
   - Dữ liệu trong Secret thường ở dạng **JSON**, được **mã hóa base64** để tăng tính bảo mật.
3. **Truy xuất Secret**: Có thể dùng:
   - Lệnh CLI:

     ```
     kubectl get secrets
     ```

   - Giao diện web của Kubernetes Dashboard hoặc IBM Cloud Console.
4. **Sử dụng thông tin đăng nhập trong ứng dụng**: Có 2 cách:
   - **Mount secret vào Pod** dưới dạng volume → tạo tệp JSON chứa đầy đủ credentials.
   - **Đưa secret vào environment variables** (ví dụ: `binding.username`, `binding.password`, `binding.apikey`).

## Ví dụ với IBM Cloud Service

Giả sử ta muốn sử dụng **Watson Tone Analyzer**:

Bước 1: Tạo service instance

- Dùng CLI hoặc IBM Cloud Catalog để tạo một instance dịch vụ.
- Ví dụ:

  ```
  ibmcloud resource service-instance-create tone-analyzer standard us-south
  ```

Bước 2: Bind service vào cluster

- Dùng lệnh:

  ```
  ibmcloud ks cluster-service-bind --cluster mycluster --namespace default --service tone-analyzer
  ```

- Sau bước này, một **Secret** được tự động sinh ra trong Kubernetes.

Bước 3: Kiểm tra Secret

```
kubectl get secrets
kubectl describe secret binding-tone-analyzer
```

Bước 4: Sử dụng trong ứng dụng

- **Mount dưới dạng file**: ứng dụng đọc file JSON trong `/etc/secrets/binding/`.
- **Sử dụng env vars**: ví dụ trong Node.js app:

  ```
  const username = process.env['binding.username'];
  const password = process.env['binding.password'];
  const apiKey   = process.env['binding.apikey'];
  ```

### Kiến trúc minh họa

- Ứng dụng trong cluster **không cần biết chi tiết về thông tin đăng nhập**.
- Binding sẽ đảm nhiệm:
  - Lấy credentials từ IBM Cloud (hoặc dịch vụ bên ngoài khác).
  - Tạo Secret trong Kubernetes.
  - Ứng dụng chỉ cần đọc Secret để kết nối đến dịch vụ.

### Lợi ích của Service Binding

- **Tự động hóa**: Credentials được cấp phát và quản lý tự động.
- **Bảo mật**: Không hard-code mật khẩu/API key, tất cả được lưu trong Kubernetes Secret.
- **Tái sử dụng**: Một service có thể được nhiều ứng dụng trong cluster cùng bind.
- **Tích hợp liền mạch**: Ứng dụng chỉ cần đọc env vars hoặc JSON file → dễ deploy CI/CD.

# Module 4: Kubernetes ecosystem

## 1. Red Hat OpenShift

### OpenShift là gì?

- **OpenShift** do Red Hat phát triển, là một **nền tảng Kubernetes sẵn sàng cho doanh nghiệp**.
- Được thiết kế cho **chiến lược đám mây lai (hybrid cloud)** và **đa đám mây (multi-cloud)**.
- Cung cấp một **nền tảng ứng dụng thống nhất**, giúp triển khai và quản lý từ môi trường phát triển đến sản xuất một cách trơn tru.
- Dựa trên:
  - **Linux** (nền tảng hệ điều hành)
  - **Container** (ứng dụng đóng gói nhẹ, di động)
  - **Automation** (tự động hóa quản trị, triển khai, mở rộng).

### Các tính năng nổi bật của OpenShift

- **Khả năng mở rộng**: chạy hàng nghìn phiên bản ứng dụng trên hàng trăm node trong vài giây.
- **Hỗ trợ hybrid cloud và edge**: quản lý linh hoạt trên nhiều hạ tầng, kể cả ở biên (edge computing).
- **Tuân thủ tiêu chuẩn mã nguồn mở**: Kubernetes, OCI (Open Container Initiative).
- **Công cụ phát triển tích hợp**: CLI (oc), tích hợp IDE, hỗ trợ đa ngôn ngữ, SDK.
- **CI/CD tích hợp**: với Jenkins và pipeline tự động hóa.
- **Nâng cấp rolling**: thực hiện upgrade “trên không” (không downtime).
- **Quản lý bảo mật**:
  - Chính sách truy cập nghiêm ngặt.
  - Quản lý rủi ro, quét lỗ hổng bảo mật, phát hiện mối đe dọa.
- **Hỗ trợ ứng dụng stateful và stateless**: tích hợp giải pháp lưu trữ doanh nghiệp.
- **Ecosystem rộng**: đối tác cung cấp dịch vụ lưu trữ, mạng, CI/CD, IDE...

### So sánh OpenShift và Kubernetes

| Khía cạnh | OpenShift | Kubernetes |
|---|---|---|
| **Loại** | Sản phẩm thương mại (Red Hat) | Dự án mã nguồn mở |
| **Triển khai** | Ít linh hoạt hơn, có quy chuẩn riêng | Triển khai được trên nhiều môi trường Linux |
| **Khả năng mở rộng dịch vụ** | Tích hợp sẵn, quản lý dễ dàng | Cần công cụ bổ sung |
| **Bảo mật** | Chính sách nghiêm ngặt, an toàn hơn | Linh hoạt, dễ cấu hình |
| **Mạng & truy cập bên ngoài** | Router object, networking out-of-the-box | Ingress object, cần plugin bên thứ ba |
| **Trải nghiệm người dùng** | Web console trực quan, dễ học | Dashboard phức tạp, ít thân thiện |
| **CI/CD** | Tích hợp Jenkins sẵn | Có thể tích hợp, nhưng cần cấu hình thêm |
| **Quản lý image** | Image streams, quản lý tốt hơn | Quản lý container image khó hơn |
| **Hạ tầng đám mây** | Có sẵn trên Azure Red Hat OpenShift, OpenShift Dedicated | Có EKS (AWS), GKE (Google), AKS (Azure) |

### Kiến trúc OpenShift

- **Lớp nền tảng**: Red Hat Enterprise Linux CoreOS (master), RHEL (worker nodes).
- **Dựa trên Kubernetes**: sử dụng `etcd` làm key-value store cho trạng thái cluster.
- **Kiến trúc microservices**:
  - REST API để quản lý đối tượng.
  - Controllers duy trì trạng thái mong muốn.
- **Các dịch vụ chính**:
  - Cluster services: giám sát tích hợp, private registry, networking.
  - Platform services: quản lý workload.
  - Application services: hỗ trợ phát triển cloud-native apps.
  - Developer services: tăng năng suất (pipeline, IDE, build automation).

### OpenShift CLI (OC)

- Công cụ CLI chính: `oc`.
- Chạy được trên Windows, Linux, Mac.
- **Bao gồm kubectl**: người dùng có thể dùng cả lệnh gốc Kubernetes và lệnh mở rộng của OpenShift.
- Các tính năng mở rộng so với kubectl:
  - Quản lý **deployment config**, **build config**.
  - **Image streams & image tags**.
  - **Routes** (truy cập dịch vụ từ bên ngoài cluster).
  - Lệnh `oc new-app`: triển khai ứng dụng từ mã nguồn hoặc image sẵn có.
  - Lệnh `oc login`: xác thực dễ dàng vào cluster.

## 2. Build

### Bản dựng là gì?

- **Bản dựng (Build)** là quá trình **biến đổi đầu vào thành một đối tượng kết quả**, ví dụ: chuyển mã nguồn thành hình ảnh container.

- Bản dựng yêu cầu một **BuildConfig** (tệp cấu hình xây dựng) để xác định:
  - **Chiến lược xây dựng** (Build Strategy)
  - **Nguồn đầu vào** (Build Source)

## Nguồn đầu vào cho bản dựng

Các nguồn đầu vào được OpenShift hỗ trợ, theo thứ tự ưu tiên:

1. **Dockerfile nội tuyến** (inline) → ưu tiên cao nhất, ghi đè Dockerfile bên ngoài.
2. **Nội dung từ hình ảnh hiện có** (image-based input)
3. **Kho Git** (Git repository)
4. **Đầu vào nhị phân hoặc cục bộ** (binary/local input)
5. **Bí mật đầu vào** (secret inputs)
6. **Hiện vật bên ngoài** (external artifacts)

Lưu ý: Có thể kết hợp nhiều nguồn đầu vào trong một bản dựng.

## Luồng hình ảnh (Image Streams)

- **Image Stream** là trừu tượng để tham chiếu các hình ảnh container trong OpenShift.
- Không chứa dữ liệu hình ảnh thực tế mà **trỏ tới các hình ảnh trong registry** (nội bộ hoặc bên ngoài).
- Một Image Stream có thể có **nhiều thẻ (tag)**, ví dụ: `dev`, `test`, `latest`.
- Khi triển khai ứng dụng, tham chiếu **thẻ luồng hình ảnh** thay vì URL/hình ảnh cứng, giúp dễ dàng cập nhật khi hình ảnh thay đổi.

## Tự động hóa bản dựng

- **Build Triggers** giúp tự động hóa bản dựng thay vì chạy thủ công:
  1. **Webhook triggers**:
     - Kích hoạt khi có commit mới, pull request hoặc webhook từ GitHub.
  2. **Image change triggers**:
     - Kích hoạt khi phiên bản mới của hình ảnh nguồn xuất hiện (ví dụ: hình ảnh Node.js cập nhật).
  3. **Config change triggers**:
     - Kích hoạt khi cấu hình BuildConfig thay đổi.

## Chiến lược xây dựng (Build Strategies)

OpenShift hỗ trợ các chiến lược phổ biến:

1. **Source-to-Image (S2I)**
   - Kết hợp **hình ảnh builder** với **mã nguồn** để tạo ra hình ảnh chạy được trong một bước.
   - Không cần Dockerfile.
   - Hỗ trợ **hình ảnh builder có sẵn** (Ruby, Node.js, Python...).
2. **Docker Strategy**
   - Sử dụng Dockerfile trong repo hoặc nội tuyến.
   - Các phương pháp: thay Dockerfile từ hình ảnh, sử dụng đường dẫn Dockerfile, biến môi trường hoặc thêm args cho build.
3. **Custom Strategy**
   - Tạo **hình ảnh builder tùy chỉnh** chứa logic build riêng.

- Có thể tạo ra các artefacts bổ sung (ví dụ: JAR, chạy CI/CD tests).
- Chỉ dành cho **quản trị viên cluster** (vì cần quyền cao).

### Quá trình build và output

- Khi build khởi động:
  1. OpenShift lấy **nguồn đầu vào**.
  2. Gọi **Docker build** (nếu Docker strategy) hoặc S2I build.
  3. Tạo **hình ảnh container mới**.
  4. Đẩy hình ảnh vào **OpenShift internal registry**.
  5. PostCommit hook (tùy chọn) có thể chạy các bước bổ sung sau build.

### Tự động hóa CI/CD

- OpenShift CI/CD pipeline:
  - Hợp nhất mã mới từ repository.
  - Xây dựng (build), kiểm tra (test), phê duyệt, triển khai (deploy) tự động.
- Mục tiêu: tăng **tự động hóa trong vòng đời container**, từ build → test → deploy.

## 3. Operator

### Toán tử (Operator) là gì?

- **Operator** là một cơ chế mở rộng Kubernetes API để **tự động hóa các tác vụ quản lý ứng dụng hoặc cụm**.
- Hoạt động như một **Custom Controller**:
  - Chạy trong pod và tương tác với **Kubernetes API server**.
  - Đóng gói, triển khai, cấu hình, và quản lý ứng dụng.
  - Thực hiện các quyết định tự động theo **thời gian thực** dựa trên trạng thái hiện tại của cụm.

### Các loại Operator

1. **Operator con người (Human Operator)**:
   - Hiểu hệ thống, biết cách triển khai và khắc phục sự cố.
2. **Operator phần mềm (Software Operator)**:
   - Nắm bắt kiến thức của người vận hành.
   - Tự động hóa quy trình, kiểm tra và nâng cấp các thành phần.

**Khác biệt với Service Broker:**

| Tiêu chí | Service Broker | Operator |
|---|---|---|
| Quy trình | Ngắn hạn | Dài hạn, theo dõi liên tục |
| Hoạt động | Chỉ cài đặt, cấu hình | Nâng cấp, mở rộng, chuyển đổi dự phòng |
| Tham số hóa | Tại thời điểm cài đặt | Liên tục, dựa trên trạng thái thực tế |

### Custom Resource Definition (CRD) và Custom Controller

- **CRD (Custom Resource Definition)**:
  - Mở rộng API Kubernetes để tạo **tài nguyên tùy chỉnh**.
  - CRDs làm cho Kubernetes trở nên **mô-đun và linh hoạt**.

- Mỗi CRD chỉ tồn tại trong cụm mà nó được cài đặt.
- Có thể truy cập bằng `kubectl` tương tự như pod, deployment.

- **Custom Controller**:
  - Đối chiếu **trạng thái thực tế của cụm** với **trạng thái mong muốn** (được định nghĩa trong CRD).
  - Kết hợp CRD + Custom Controller = **mẫu Operator** (Operator Pattern).

### Operator SDK & Framework

- **Operator Framework**: bộ công cụ mã nguồn mở để:
  - Mã hóa, kiểm tra, đóng gói, phân phối và cập nhật Operator.
- **Operator SDK**: hỗ trợ phát triển bằng **Helm, Go hoặc Ansible**.
- **Operator Lifecycle Manager (OLM)**:
  - Quản lý cài đặt, nâng cấp và kiểm soát truy cập (RBAC) của Operators.
  - Lưu trữ thông tin CRD, Cluster Service Version (CSV) và metadata.
- **Operator Hub**: bảng điều khiển web để **tìm và cài đặt Operators** dễ dàng.

### Mục đích và khả năng của Operators

- **Triển khai ứng dụng**: CRD tạo các tài nguyên tùy chỉnh, Operator quản lý deployment, service, storage...
- **Mở rộng ứng dụng**: tự động thêm nhiều pod theo nhu cầu.
- **Tự động hóa các tác vụ**: backup, restore, scaling, cấu hình, patching...
- **Đảm bảo trạng thái mong muốn**: Operator liên tục điều chỉnh trạng thái thực tế theo CRD.

### Mô hình trưởng thành (Operator Maturity Model)

- Xác định các giai đoạn trưởng thành cho **hoạt động hàng ngày**:
  1. **Cài đặt cơ bản**
  2. **Vận hành thủ công**
  3. **Một số tự động hóa**
  4. **Hoạt động tự động hoàn toàn**
- Mức trưởng thành cao hơn cho phép tự động hóa **nâng cấp, chuyển đổi dự phòng, scaling** và tích hợp CI/CD.

### Ví dụ về Operators

- **Red Hat Certified Operator**: cài đặt và quản lý trên OpenShift.
- **Community Operator**: từ cộng đồng nguồn mở, không chính thức hỗ trợ Red Hat.
- **User-defined Operator**: tự tạo để quản lý ứng dụng hoặc dịch vụ đặc thù.
- **Service Mesh (Istio)**: Operator triển khai và quản lý dịch vụ mạng trong cụm.

## 4. Istio

### Service Mesh là gì?

- **Service Mesh** là một lớp phần mềm chuyên dụng giúp **giao tiếp giữa các dịch vụ trở nên an toàn và đáng tin cậy**.
- Cung cấp các khả năng chính:
  1. **Quản lý lưu lượng**: kiểm soát luồng giữa các dịch vụ.

2. **Bảo mật**: mã hóa lưu lượng, xác thực và ủy quyền dịch vụ.
3. **Khả năng quan sát**: giám sát, logging, tracing các luồng giao tiếp.

- **Istio** là một service mesh độc lập với nền tảng, thường chạy trên Kubernetes.

## Bốn khái niệm mà Istio hỗ trợ

1. **Kết nối (Connectivity)**:
   - Kiểm soát thông minh lưu lượng giữa các dịch vụ (ví dụ: Canary, thử nghiệm A/B).
2. **Bảo mật (Security)**:
   - Mã hóa lưu lượng giữa các dịch vụ, xác thực, ủy quyền, thực thi chính sách.
3. **Thực thi (Policy Enforcement)**:
   - Kiểm soát truy cập, giới hạn tỷ lệ, hạn ngạch.
4. **Khả năng quan sát (Observability)**:
   - Giám sát luồng giao tiếp, theo dõi phụ thuộc giữa các dịch vụ, số liệu như độ trễ và lỗi.

## Kiến trúc Istio

- **Hai thành phần chính**:
  1. **Control Plane (Mặt phẳng điều khiển)**:
     - Cấu hình trạng thái mong muốn.
     - Cập nhật các proxy khi môi trường thay đổi.
  2. **Data Plane (Mặt phẳng dữ liệu)**:
     - Proxy **Envoy** chặn tất cả lưu lượng giữa các dịch vụ.
     - Thực hiện routing, retry, failover, và các chính sách bảo mật.

## Lợi ích của Istio với Microservices

- Cho phép **thử nghiệm Canary và A/B** mà không thay đổi code ứng dụng.
- Cải thiện **bảo mật giao tiếp dịch vụ** với TLS và kiểm soát truy cập.
- Hỗ trợ **tự động retry, ngắt mạch (circuit breaker)** để tránh lỗi lan truyền.
- Dễ dàng **quan sát và tối ưu hóa luồng dịch vụ**.
- Cho phép **triển khai và nâng cấp dịch vụ riêng lẻ** mà không ảnh hưởng toàn bộ ứng dụng.

## Thách thức của Microservices và cách Istio giải quyết

- **Giao tiếp phức tạp**: nhiều dịch vụ tương tác dẫn đến lỗi xếp tầng.
- **Bảo mật**: cần mã hóa và kiểm soát quyền truy cập.
- **Triển khai thử nghiệm**: cần Canary, A/B testing.
- Istio giải quyết bằng **routing thông minh, bảo mật, policy enforcement và observability**.

## Khả năng giám sát của Istio

**Bốn số liệu giám sát dịch vụ cơ bản**:

1. **Độ trễ (Latency)**: thời gian yêu cầu/response.
2. **Lưu lượng (Traffic)**: số lượng yêu cầu.
3. **Lỗi (Errors)**: số lượng lỗi hoặc thất bại.
4. **Độ bão hòa (Saturation)**: mức độ sử dụng tài nguyên, tắc nghẽn.

### Các tính năng nâng cao của Istio

- **Routing linh hoạt**: chuyển dần lưu lượng giữa các phiên bản dịch vụ.
- **Mã hóa end-to-end** giữa các microservices.
- **Kiểm soát truy cập dịch vụ**: đảm bảo dịch vụ chỉ nói chuyện với dịch vụ cần thiết.
- **Hỗ trợ nhiều giao thức**: HTTP, TCP, gRPC, WebSocket.