Addressing production-ready distributed systems through Kubernetes

A practical overview

Mattia Matteini

Alma Mater Studiorum - University of Bologna

October 13, 2025

Table of Contents

1. Motivations
2. Premises
3. Kubernetes introduction
   - Differences with Docker Swarm
   - Key features
   - Cluster Architecture
   - Objects
4. A practical example
   - Development with Docker Compose
   - Migrating to Kubernetes
   - Stress testing the system
5. System monitoring
6. References

Motivations

1. Containerization frameworks, such as Docker, opened the doors to new ways to develop and deploy distributed systems
2. Most of the software systems nowadays are distributed (and complex)
3. Non-functional requirements (quality attributes) acquire more importance in the design of these systems
4. Kubernetes is now the standard de facto for container orchestration.

Table of Contents

1. Motivations
2. Premises
3. Kubernetes introduction
   - Differences with Docker Swarm
   - Key features
   - Cluster Architecture
   - Objects
4. A practical example
   - Development with Docker Compose
   - Migrating to Kubernetes
   - Stress testing the system
5. System monitoring
6. References

Premises

Containerization

- A lightweight form of virtualization
- It replaced the traditional Virtual Machines in many deployment scenarios
- Provides process and filesystem isolation exploiting Linux kernel
- Popular containerization platform: Docker [2]

To deepen the topic, see Giovanni Ciatto's lecture on Virtuale.

Table of Contents

1. Motivations
2. Premises
3. Kubernetes introduction
   - Differences with Docker Swarm
   - Key features
   - Cluster Architecture
   - Objects
4. A practical example
   - Development with Docker Compose
   - Migrating to Kubernetes
   - Stress testing the system
5. System monitoring
6. References

Kubernetes introduction

What Kubernetes is

A portable, extensible, open source platform for managing containerized workloads and services [6]

What Kubernetes is not

- A containerization platform (like Docker)
- A Platform as a Service (like Heroku [4])
- A cloud provider (but works with them)
- A deployment tool (CI/CD is defined by organizations)
- A logging/monitoring tool (but offers integrations for them)

Differences with Docker Swarm I

On Docker Swarm:

- Native clustering and orchestration tool for Docker
- Simpler and easier to set up compared to Kubernetes
- Suitable for smaller clusters and less complex applications
- Users can manually handle resource allocation and scaling
- Environment configuration is managed through the *docker-compose.yml* file.
- Simple access control based on TLS

Differences with Docker Swarm II

On Kubernetes:

- Open-source container orchestration platform
- More complex and feature-rich compared to Docker Swarm
- Ideal for larger clusters and more complex applications
- Automatically handles scaling, load balancing and failover
- Environment configuration is managed through YAML files
- Advanced access control with Role-Based Access Control (RBAC)

Key features I

Immutability

- Kubernetes resources cannot be changed after they are created
- If a change is needed, the resource is deleted and a new one is created
- Containers are meant to be ephemeral (effimeri (temporanei)) and stateless
  - deleting and recreating containers is a standard part of their lifecycle
- Ensures consistency and reliability in the cluster
- Supports declarative configuration
  - changes produce new desired states rather than mutating existing ones

Key features II

Declarative configuration

Aderisce al principio

- Adheres to the principle *"everything is an object"*
- Several kinds of objects are available to shape the production environment
- Configuration files are written in yaml (or json)
- External declarative tool, named `kubectl`, to manage the environment.
  - Example of creating a Kubernetes resource:
    `kubectl create -f configuration-file.yaml`

Helps to improve *Configurability* and *Maintainability*.

Key features III

Autoscaling

Kubernetes supports automatic scaling of applications based on resource usage or other metrics. There are two main types of autoscaling:

1. **Horizontal Scaling**: scales up and down replicas of a resource
2. **Vertical Scaling**: adjusts the resource available to a container (e.g., CPU, memory)

Helps to improve *Availability* and *Scalability* of applications while optimizing resource usage.

Key features IV

Self-healing

- Kubernetes automatically replaces failed containers
- If a service on a node is running with a volume attached, and the node fails, Kubernetes can reattach the volume to a new instance on a different node
- If a container behind a service fails, Kubernetes automatically removes its route and redirects traffic to other healthy instances of same service

Helps to improve *Recoverability* and *Reliability*.

Key features V

Container Runtime Interface (CRI)

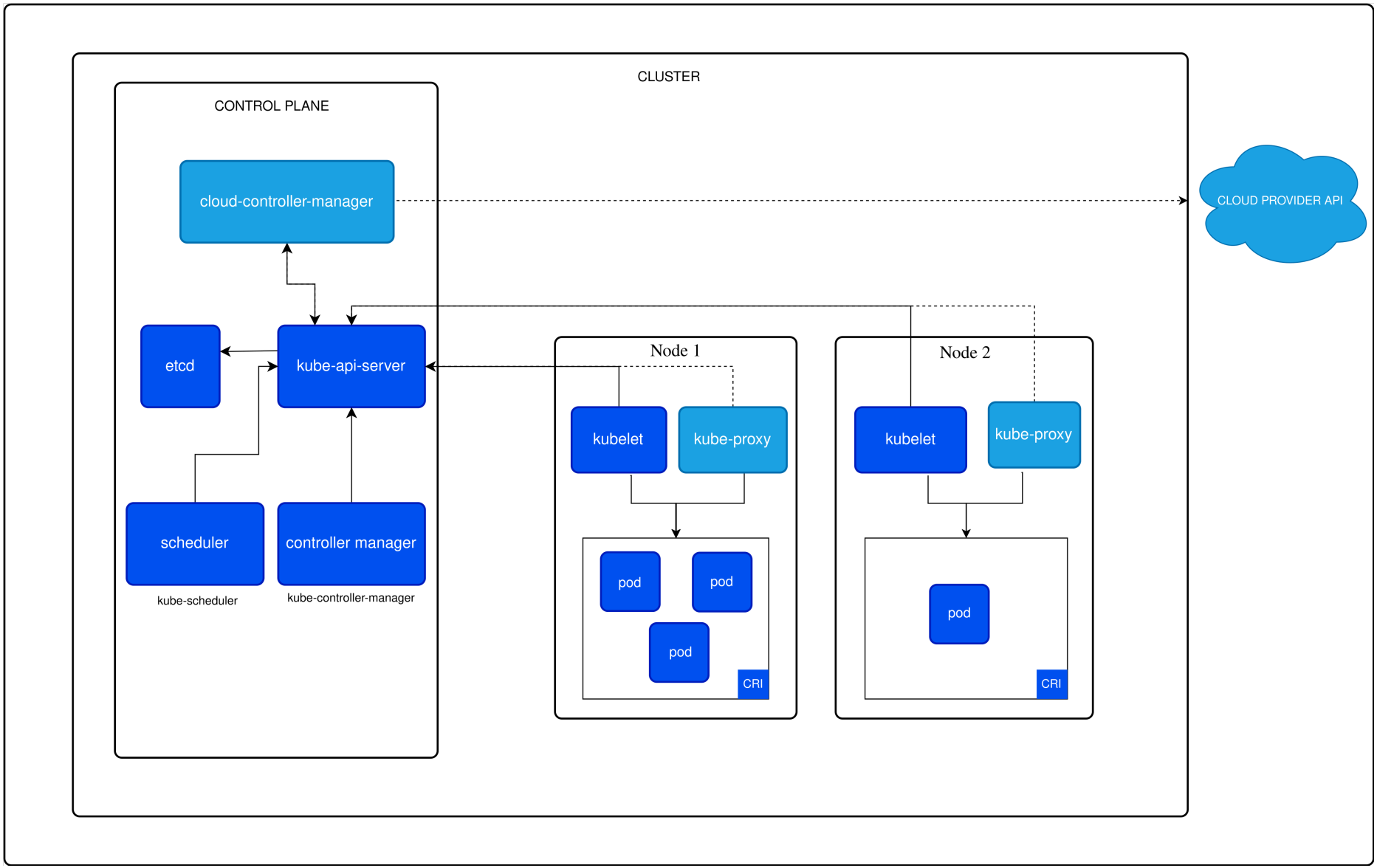
- The CRI is a plugin interface which enables Kubernetes to use a wide variety of container runtimes
- Such container runtime must be working on each cluster node
- This means that Kubernetes is not tied to Docker only.
  - The other container runtimes currently supported are: containerd, CRI-O, and Mirantis Container Runtime

Cluster Architecture I

A Kubernetes cluster consists of a control plane plus a set of worker machines, called nodes.

- **Control Plane**: manages the Kubernetes cluster.
  - *kube-apiserver*: component that exposes the Kubernetes HTTP API
  - *etcd*: a key-value database used to store cluster metadata
  - *kube-scheduler*: component that watches for newly created services with no assigned node, and selects a node for them to run on
  - *kube-controller-manager*: component that runs controller processes
  - *cloud-controller-manager*: component that embeds cloud-specific control logic
- **Worker Node**: machine running containerized applications and the following components:
  - *kubelet*: ensures that containers are running
  - *kube-proxy*: maintains network rules on nodes
  - *container runtime*: software responsible for running containers (e.g., Docker)

Cluster Architecture II



Objects I

- Objects are *persistent entities* in the Kubernetes system
- Kubernetes uses these entities to represent the *state of the cluster*
  - What containerized applications are running (and on which nodes)
  - The resources available to those applications
  - Policies for application behavior, such as restarts and upgrades
- A Kubernetes object is a "record of intent"
  - When the object is created, the Kubernetes system will constantly work to ensure that the object exists
  - Creation of an object tells the Kubernetes system how the cluster should look like — i.e. its desired state

Objects II

Object properties

Almost every Kubernetes object includes two nested object fields

- **Spec**: provides a description of the characteristics you want the resource to have (the *desired state*)
- **Status**: describes the *current state* of the object, supplied and updated by the Kubernetes system and its components
  - Kubernetes control plane continually manages every object's actual state to match the desired state you supplied

Objects III

Creating objects

- Creating an object means define the *object spec* and some basic information about the object (such as a name)
- Usually, this is done providing to `kubectl` a configuration file, known as *manifest*, written by convention in YAML
- Required fields in the manifest:
  - `apiVersion`: the version of the Kubernetes API to use to
  - `kind`: the type of object being created
  - `metadata`: data that helps uniquely identify the object, such as name and UID
  - `spec`: the desired state of the object
    - Note that the format of the object `spec` is different for every Kubernetes object, and contains nested fields specific to that object

Objects IV

The following manifest file creates a Kubernetes object of kind `Pod`, which contains a single container running the `nginx` image

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx:1.14.2
      ports:
        - containerPort: 80
```

To apply this configuration:
`kubectl apply -f <your-manifest-file>`,

Pods I

- Pods are the *smallest deployable units*
- They represent a group of one or more containers, with shared storage and network resources
- The *shared context* of a Pod is a set of Linux namespaces, cgroups, and potentially other facets of isolation (just like in containers)
- Two ways to use Pods:
  - Running a single container: the most common use case, Pod is used as a wrapper around a single container
  - Running multiple containers: Pod is used to encapsulate an application composed of multiple co-located tightly coupled containers

Pods II

Running multiple containers in a Pod

- For relatively advanced use case
- Useful to implement patterns like Sidecar, Ambassador, and Adapter [1] (non-exhaustive list)
- Containers in a Pod share:
  - IP address and ports
  - Hostname
  - Storage volumes
  - Process identifiers (PIDs)

Pods III

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: node-app
  labels:
    app: node
spec:
  containers:
    - name: node-container
      image: node:18
      command: ["node", "-e", "console.log('Hello World!')"]
      resources:
        requests:
          memory: "128Mi"
          cpu: "100m"
        limits:
          memory: "256Mi"
          cpu: "500m"
      ports:
        - containerPort: 3000
          protocol: TCP
```

Pods IV

About pod configuration

- It is possible to specify
  - the `image` to use for the container
  - the `command` to run in the container
  - the `port` exposed by the container
  - the network `protocol` used (e.g., TCP, UDP)
- The `resources` field manages hardware resources
  - `requests`: the amount of CPU/memory that Kubernetes guarantees to the container
  - `limits`: the maximum amount of CPU/memory that the container can use
  - CPU resources are expressed in millicpu units (0.1 = 10% = 100m)
  - Memory resources are expressed in bytes (122Mi = 128MB)

ReplicaSets

- The object that manages the number of *replicas* of a Pod
- Allows to scale the replicas of Pods up and down
- It is often used to guarantee the availability of a specified number of identical Pods
- When a ReplicaSet needs to create new Pods, it uses its Pod template
- Usually not directly managed by the user, but through a Deployment

Deployments I

- Pods (and ReplicaSets) are not directly managed by the user, but through higher-level objects called `Deployments`
- A Deployment manages a set of Pods to run an application workload
- They provide *declarative updates* for Pods and ReplicaSets like Rollout (change from current state to the desired state) and Rollback
  - for example changing the image version of a container
  - avoiding downtime during the update process
- Used to manage the release of a new version of the application

Deployments II

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the
      template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

Deployments III

About deployment configuration

- The `replicas` field specifies the number of desired Pod replicas
- The `selector` field defines how the Deployment finds which Pods to manage
- The `template` field contains the Pod specification that the Deployment uses to create new Pods

Deployments IV

Deployments are not the only solution

- `Job`: for running short-lived, one-off tasks
- `CronJob`: meant for performing regular scheduled actions such as backups, report generation, and so on
- `StatefulSet`: manages Pods like a Deployment, but mantains a sticky identity for each of its Pods
- `DaemonSet`: ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them, and as nodes are removed from the cluster, those Pods are garbage collected.

Services I

- An abstraction which defines a logical set of Pods and a policy by which to access them
- The *entry-point* of the application
- They redirect requests to the Pods in the cluster nodes
- Pods are ephemeral, so they can be moved between nodes, and their IP address is subject to changes
  - A Service enables a *stable endpoint* (IP address and DNS name) to access a set of Pods exploiting label selectors
- Different types of Services:
  - `ClusterIP` (default): exposes the service on a cluster-internal IP
  - `NodePort`: exposes the service on each Node's IP at a static port
  - `LoadBalancer`: exposes the service externally using a load balancer provided by a cloud provider (or using MetalLB [7] on-premise)

Services II

```yaml
apiVersion: v1
kind: Service
metadata:
  name: node-service
  labels:
    app: node
spec:
  selector:
    app: node            # Must match Pod label
  ports:
    - protocol: TCP
      port: 80           # Service port (client-facing)
      targetPort: 3000 # Container port (inside the Pod)
  type: ClusterIP
```

Table of Contents

1. Motivations
2. Premises
3. Kubernetes introduction
   - Differences with Docker Swarm
   - Key features
   - Cluster Architecture
   - Objects
4. A practical example
   - Development with Docker Compose
   - Migrating to Kubernetes
   - Stress testing the system
5. System monitoring
6. References

A practical example

Context

A simple web application that counts the number of visits composed of:

- A backend service (Python + Flask)
- A in-memory key-value store (Redis [10])

The repository is available on Github:
https://github.com/Mala1180/kubernetes-hpa-example

What we are interested in

1. Development phase using Docker Compose
2. Migration to Kubernetes
3. Monitoring using Prometheus and Grafana

Development with Docker Compose I

A simple hit counter implementation using Flask and Redis

app.py:

```
1  app = Flask(__name__)
2
3  # Connect to Redis
4  redis_host = os.environ.get("REDIS_HOST", "redis")
5  redis_port = int(os.environ.get("REDIS_PORT", 6379))
6  cache = redis.Redis(host=redis_host, port=redis_port)
7
8
9  @app.route("/")
10 def hello():
11     # Increment hit counter
12     hits = cache.incr("hits")
13     # Get the Pod hostname
14     hostname = socket.gethostname()
15     return (
16         f"Hello! This page has been visited {hits} times. "
17         f"<br>Served by pod: <b>{hostname}</b>\n"
18     )
```

Development with Docker Compose II

```
19
20
21  @app.route("/busy-wait")
22  def busy_wait():
23      # Not interesting busy waiting task
24      ...
25      return "Finished busy-waiting.\n"
26
27  if __name__ == "__main__":
28      app.run(host="0.0.0.0", port=3000, debug=True)
```

Development with Docker Compose III

Let's containerize the application using Docker and Docker Compose to ease the development phase

facilitare

```
1  FROM python:3.12-slim
2
3  WORKDIR /app
4
5  COPY requirements.txt .
6  RUN pip install --no-cache-dir -r requirements.txt
7
8  COPY requirements.txt .
9  COPY app.py .
10 COPY pyproject.toml .
11 COPY poetry.lock .
12
13 RUN pip install --upgrade pip
14 RUN pip install --no-cache-dir -r requirements.txt
15 RUN poetry install --no-root
16
17 EXPOSE 3000
18 CMD ["poetry", "run", "python", "app.py"]
```

Development with Docker Compose IV

```
1  services:
2    backend:
3      image: [dockerhub-username]/kubernetes-example-backend
           :latest # username is for push on DockerHub
4      build: .
5      ports:
6        - "3000:3000"
7      environment:
8        - REDIS_HOST=redis
9        - REDIS_PORT=6379
10     depends_on:
11       - redis
12
13   redis:
14     image: redis:7-alpine
```

Development with Docker Compose V

To build and run the application:

- `docker compose up -d --build`
- visit http://localhost:3000
- you should see something like this:

Hello! This page has been visited 1 times.
Served by pod: **f7b798038e89**

Migrating to Kubernetes I

At this point, let's migrate to Kubernetes

Mapping to Kubernetes objects

For each service in the docker compose:

- Define a Deployment object
- Define a Service object

In addition, let's define a Horizontal Pod Autoscaler object to automatically scale the backend service

Migrating to Kubernetes II

backend-deployment.yaml:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
  labels:
    app: backend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
        - name: backend
          image: mala1180/kubernetes-example-backend:latest
```

Migrating to Kubernetes III

```
20              imagePullPolicy: Always
21              ports:
22                - containerPort: 3000
23              env:
24                - name: REDIS_HOST
25                  value: "redis"
26                - name: REDIS_PORT
27                  value: "6379"
28              resources:
29                requests:
30                  memory: "128Mi"
31                  cpu: "100m"
32                limits:
33                  memory: "256Mi"
34                  cpu: "200m"
```

Migrating to Kubernetes IV

`backend-service.yaml`:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: backend
  labels:
    app: backend
spec:
  type: NodePort
  ports:
    - port: 3000
      targetPort: 3000
      protocol: TCP
  selector:
    app: backend
```

And similarly, for Redis service

Migrating to Kubernetes V

Finally, the Horizontal Pod Autoscaler triggered if the CPU usage goes above 50% or the memory usage goes above 70%

`backend-hpa.yaml`:

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: backend-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: backend
  minReplicas: 1
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
```

Migrating to Kubernetes VI

```
17            type: Utilization
18            averageUtilization: 50
19      - type: Resource
20        resource:
21          name: memory
22          target:
23            type: Utilization
24            averageUtilization: 70
```

Migrating to Kubernetes VII

Kompose

Kompose [5] is a useful tool to save time in converting Docker Compose files to Kubernetes configuration files. It supports Kompose-specific labels within the compose.yml file to explicitly define the behavior of the generated resources upon conversion, such as Services, Deployments, etc. For example:

- adding labels to the docker-compose.yml file

```
services:
  backend:
    ...
    labels:
      kompose.service.type: nodeport
      kompose.controller.type: deployment
      kompose.image-pull-policy: always
```

- `kompose convert -f docker-compose.yml -o k8s`

Migrating to Kubernetes VIII

At this point:

- start Minikube [8] to deploy the system locally

```
minikube start
```

- enable the metrics server to use the Horizontal Pod Autoscaler

```
minikube addons enable metrics-server
```

- apply the Kubernetes manifests

```
kubectl apply -f k8s
```

- access the web page

```
minikube service backend
kubectl port-forward svc/backend 8080:3000
```

- visit http://localhost:8080
- monitor the status of the system through the dashboard

```
minikube dashboard
```

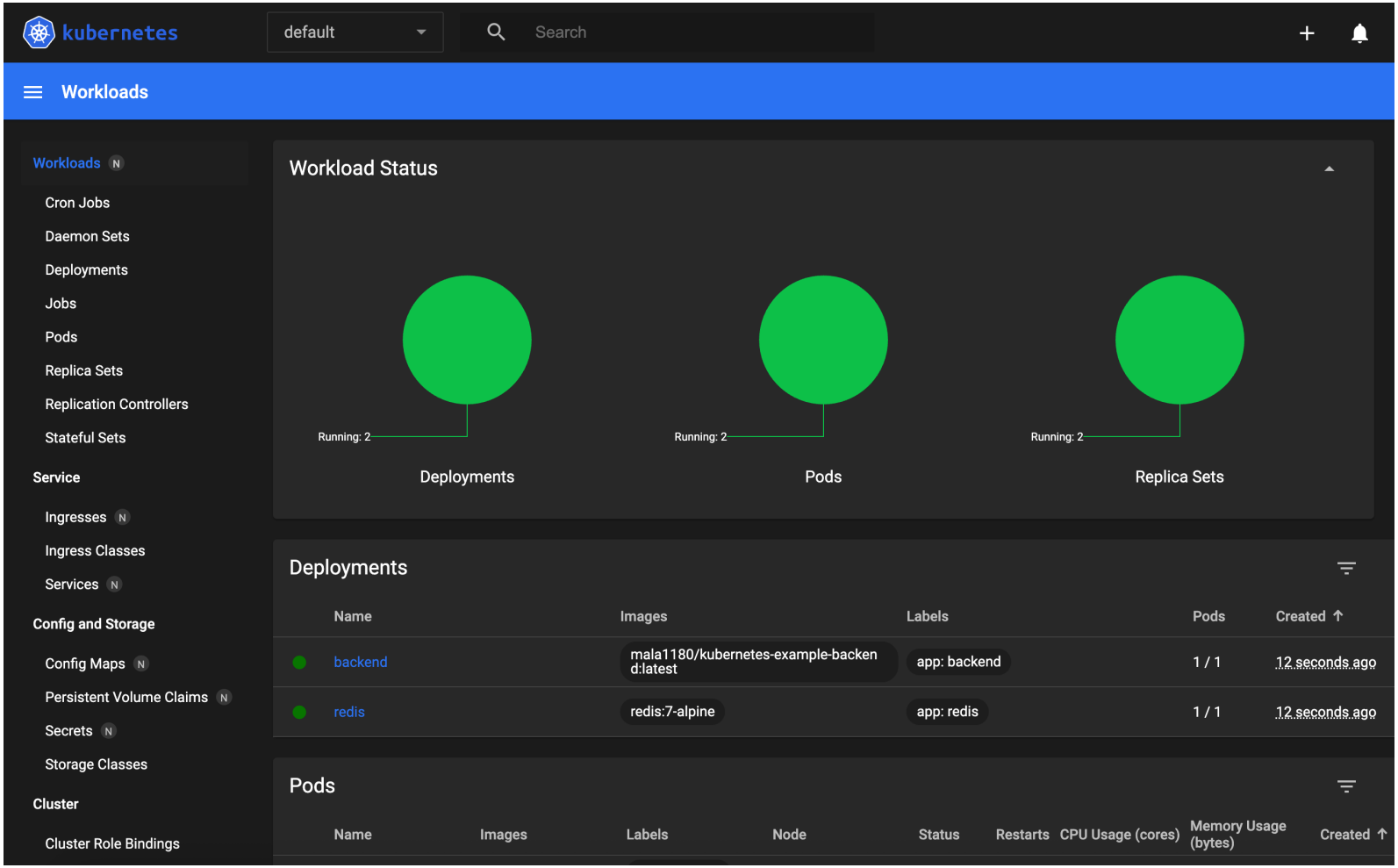
Stress testing the system I

In order to test the autoscaling capabilities of the system, we can run a script that generates load on the backend service

- run the script `stress-test.sh`
- immediately, the web page should become unresponsive because of the high load of requests
- in few seconds, the Horizontal Pod Autoscaler should kick in and start new pod replicas
- after a while, the web page should become responsive again, showing that there are multiple pods serving requests (with different pod ids)

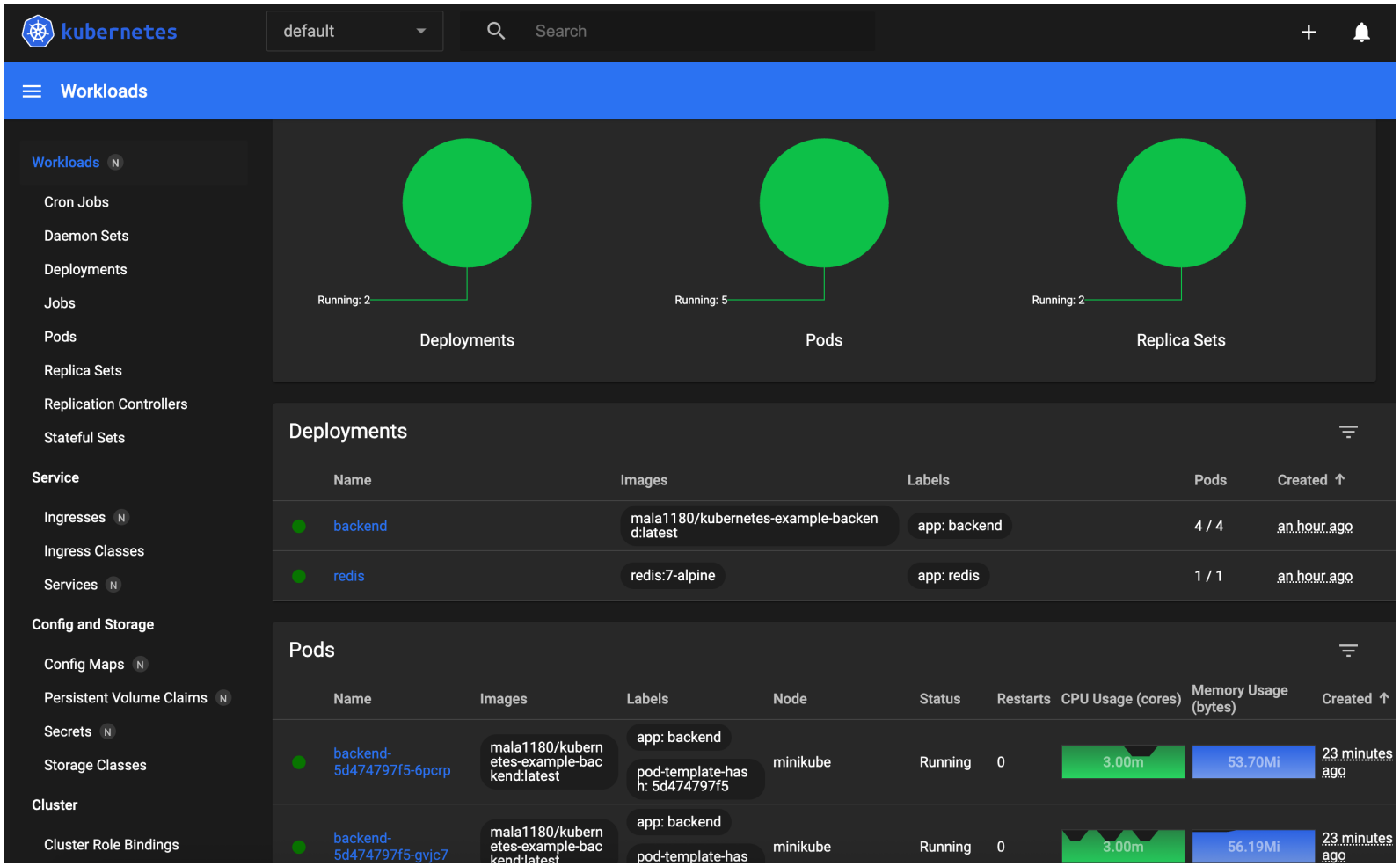
Stress testing the system II

Dashboard view when the system is under normal load:



Stress testing the system III

Dashboard view when the system is under stress test:



Stress testing the system IV

When replicas are created, the web page comes back to be responsive. The Pod id in the page shows that there are multiple pods serving requests.

Hello! This page has been visited 40 times.
Served by pod: **web-6d86bf6f4f-8ppcq**

Hello! This page has been visited 49 times.
Served by pod: **web-6d86bf6f4f-w9z4f**

Hello! This page has been visited 53 times.
Served by pod: **web-6d86bf6f4f-mqr57**

Table of Contents

1. Motivations
2. Premises
3. Kubernetes introduction
   - Differences with Docker Swarm
   - Key features
   - Cluster Architecture
   - Objects
4. A practical example
   - Development with Docker Compose
   - Migrating to Kubernetes
   - Stress testing the system
5. System monitoring
6. References

System monitoring I

Observability

- So far we mainly focused on Scalability, Availability and Reliability
- However, another important QA is *Observability*
- A production system eventually will go down for any reason
- An observable system allows understanding what is happened/happening

System monitoring II

Monitoring

- Some monitoring tools help to achieve observability.
- A monitoring system can:
  - Collect and show metrics
  - Alert the system administrator
  - Log events
- The most common are:
  - **Prometheus**: open-source systems monitoring and alerting toolkit [9]
  - **Grafana**: open-source web interface for analytics and monitoring [3]

System monitoring III

Grafana dashboard showing system metrics collected by Prometheus during the stress test

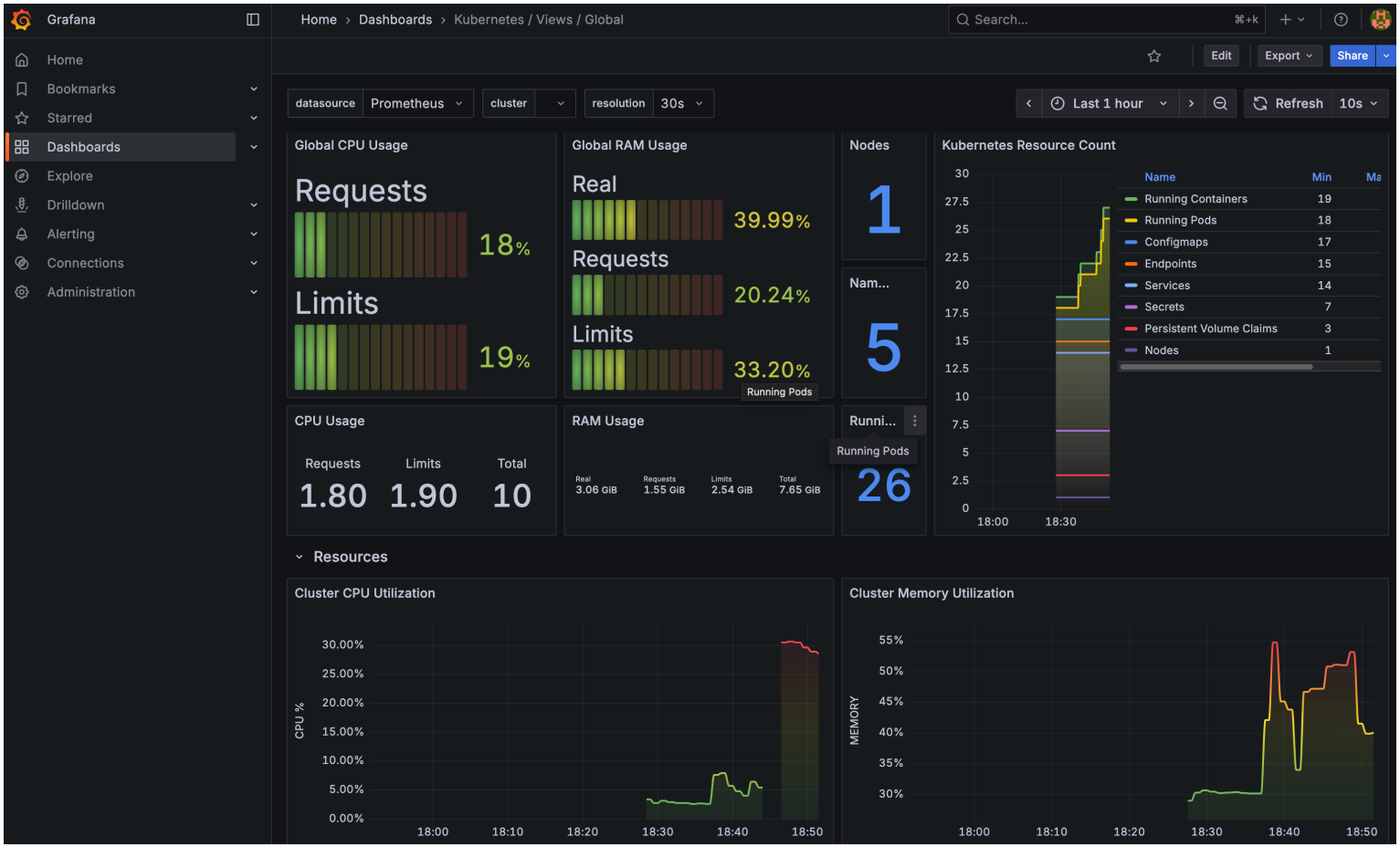


Table of Contents

1. Motivations
2. Premises
3. Kubernetes introduction
   - Differences with Docker Swarm
   - Key features
   - Cluster Architecture
   - Objects
4. A practical example
   - Development with Docker Compose
   - Migrating to Kubernetes
   - Stress testing the system
5. System monitoring
6. References

References I

[1] Brendan Burns. *Designing distributed systems.* " O'Reilly Media, Inc.", 2024.

[2] *Docker*. URL: https://www.docker.com/.

[3] *Grafana*. URL: https://grafana.com/.

[4] *Heroku*. URL: https://www.heroku.com/.

[5] *Kompose*. URL: https://kompose.io.

[6] *Kubernetes*. URL: https://kubernetes.io/docs/concepts/overview/.

[7] *MetalLB*. URL: https://metallb.io/.

[8] *Minikube*. URL: https://minikube.sigs.k8s.io/docs/.

[9] *Prometheus*. URL: https://prometheus.io/.

[10] *Redis*. URL: https://redis.io/.