ITI8801 · Cloud Architectures and DevOps

Kubernetes: running containers on a cluster

Siim Rebane · TalTech · Autumn 2026

The problem

Assignment 5 ran three containers from one `compose.yaml` on one VM. Compose starts them and restarts them when they exit. It does not handle:

Situation	With Compose on one VM
the VM stops	every container stops with it
more load than one VM can carry	a second VM is a second, separate Compose project
a new image version	`docker compose up -d` stops the old container, then starts the new one: the service is down in between
deciding which container runs on which machine	done by a person, by hand

Running containers on many machines needs a program that places them, watches them and replaces them.

An orchestrator

A **container orchestrator** runs containers on a group of machines. You describe the result you want; the orchestrator keeps it true:

- places each container on a machine that has room for it
- starts a replacement when a container or a whole machine fails
- runs as many copies as you asked for, and changes the number when you change it
- replaces old versions with new ones a few at a time
- gives a group of copies one stable name and address

It is the same idea as Terraform's desired state, with one difference: Terraform compares and changes when you run `apply`; an orchestrator compares all the time and corrects on its own.

Ways to run containers

Way	Examples	You run	The provider runs
one machine	Docker, Docker Compose	the machine, Docker, the containers	the VM, if it is a cloud VM
a simple orchestrator	Docker Swarm, HashiCorp Nomad	the machines, the orchestrator, the containers	the VMs
Kubernetes, run by you	k3s and other installers	the machines, the orchestrator's own programs and their upgrades, the containers	the VMs
Kubernetes, run by a cloud	Azure, AWS and Google each sell one	the machines' size and number, the containers	the orchestrator's own programs, and the VMs
managed container services	Azure Container Apps, AWS ECS with Fargate, Google Cloud Run	an image and its settings	everything that runs it

Top to bottom: less to run yourself, less control over how it runs, and more of the setup specific to one provider.

Docker Swarm

Docker Swarm is the orchestrator built into Docker Engine: `docker swarm` commands turn machines that run Docker into one cluster, a **swarm**.

Term	Is
node	one machine in the swarm
manager	a node that keeps the swarm's state and decides where containers run; managers agree by majority, so three survive the loss of one and five the loss of two
worker	a node that only runs containers
service	what you ask for: an image, a number of copies, ports, networks
task	one copy of a service, run as one container on one node
stack	a group of services deployed together from a Compose file

HashiCorp Nomad is another orchestrator of this size: one program, jobs written in HCL, the language of Terraform, and it runs plain programs as well as containers.

Starting a swarm

```
$ docker swarm init --advertise-addr 192.168.42.244
Swarm initialized: current node (r27z3jcgvjzb9r6sk4eidtwpa) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-… 192.168.42.244:2377

$ docker node ls
ID                            HOSTNAME   STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
r27z3jcgvjzb9r6sk4eidtwpa *   vm-79      Ready     Active         Leader           29.1.3
```

- `--advertise-addr` : the address other nodes use to reach this one
- `docker swarm join --token … <manager>:2377` on another machine with Docker makes it a worker; the token is the secret that lets it in
- `Leader` : the manager that currently decides; the others follow it

A service

```
$ docker service create --name web --replicas 2 --publish 8088:80 nginx:1.29-alpine
$ docker service ls
ID             NAME      MODE         REPLICAS   IMAGE                PORTS
zwdm5z5ohfa3   web       replicated   2/2        nginx:1.29-alpine   *:8088->80/tcp
$ docker service ps web
ID             NAME      IMAGE                NODE      DESIRED STATE   CURRENT STATE
431sebzdtqnm   web.1     nginx:1.29-alpine   vm-79     Running         Running 8 seconds ago
pzmhr7qrsyfp   web.2     nginx:1.29-alpine   vm-79     Running         Running 8 seconds ago
```

Part	Means
`--replicas 2`	two tasks
`--publish 8088:80`	port 8088 on **every** node of the swarm leads to port 80 of one of the tasks, wherever it runs
`docker service ps web`	the tasks, and the node each one runs on
`DESIRED STATE`, `CURRENT STATE`	what the manager wants for the task, and what the task is doing now

Each task is an ordinary container on its node.

A task stops: the swarm starts another

The container of task `web.1`, found with `docker ps` (`--format`: print only these two columns) and removed by force, as a crash would end it. A task's container is named `<service>.<slot>.<task ID>`:

```
$ docker ps --format "{{.ID}}  {{.Names}}"
ea649036f983  web.1.431sebzdtqnmbw64i0fmwvw78
a1fbeab0f5e1  web.2.pzmhr7qrsyfph27zf1ls4livh
$ docker rm -f ea649036f983
$ docker service ps web
ID             NAME        IMAGE               NODE    DESIRED STATE   CURRENT STATE            ERROR
o9e6zzfhtrfr   web.1       nginx:1.29-alpine   vm-79   Running         Running 2 seconds ago
431sebzdtqnm    \_ web.1   nginx:1.29-alpine   vm-79   Shutdown        Failed 8 seconds ago     "task: non-zero exit (137)"
pzmhr7qrsyfp   web.2       nginx:1.29-alpine   vm-79   Running         Running 16 seconds ago
```

In the output	Means
`web.1` with a new ID, `o9e6zzfhtrfr`	the manager saw one task running where the service asks for two, and started a new task in slot `web.1`
the line `\_ web.1`, ID `431sebzdtqnm`	the removed task, kept as history under its slot
its `DESIRED STATE` `Shutdown`, `CURRENT STATE` `Failed`	the manager no longer wants it; it ended by failing
its `ERROR` `non-zero exit (137)`	the container was killed by signal 9, `SIGKILL`: 128 + 9. `docker rm -f` ends a running container that way

A node that leaves the swarm is handled the same way: its tasks start again on the other nodes.

Scaling

```
$ kubectl scale deployment api --replicas=3
deployment.apps/api scaled
$ kubectl get deployment api
NAME   READY   UP-TO-DATE   AVAILABLE   AGE
api    3/3     3            3           52s
```

- `scale web=3` adds one task, `web.3`; `web.1` and `web.2` keep running untouched. Scaling down removes tasks the same way
- `--filter desired-state=running`: only the tasks that should be running, without the history

Updating

A new image version for the service:

```
$ docker service update --image nginx:1.30-alpine web
$ docker service ps web
ID             NAME        IMAGE                NODE    DESIRED STATE   CURRENT STATE
aoqu6tonygkt   web.1       nginx:1.30-alpine    vm-79   Running         Running 16 seconds ago
o9e6zzfhtrfr    \_ web.1   nginx:1.29-alpine    vm-79   Shutdown        Shutdown 18 seconds ago
431sebzdtqnm    \_ web.1   nginx:1.29-alpine    vm-79   Shutdown        Failed 39 seconds ago
i9b7pnsqh0cw   web.2       nginx:1.30-alpine    vm-79   Running         Running 8 seconds ago
pzmhr7qrsyfp    \_ web.2   nginx:1.29-alpine    vm-79   Shutdown        Shutdown 8 seconds ago
tyvsr6x1r8b6   web.3       nginx:1.30-alpine    vm-79   Running         Running 12 seconds ago
69ivihiwbvw9    \_ web.3   nginx:1.29-alpine    vm-79   Shutdown        Shutdown 13 seconds ago
```

- a running container cannot change its image: every task is replaced by a new one with `nginx:1.30-alpine`
- one task at a time, so the others keep serving: `web.1` 16 s ago, `web.3` 12 s ago, `web.2` 8 s ago
- the same happens for any change to what the tasks run: environment variables, command, ports

A Compose file as a stack

Lecture 5's `compose.yaml` for Notes, with two changes, saved as `stack.yaml`. The `db` service is unchanged: `postgres:18`, the volume `pgdata`, the healthcheck.

```yaml
  api:
    image: ghcr.io/siimrebane/iti8801-notes:a4     # was: build: . — a swarm pulls images, it builds nothing
    environment:
      MODE: api
      DATABASE_URL: postgres://postgres:${DB_PASSWORD}@db:5432/notes?sslmode=disable
    networks: [frontend, backend]
    deploy:
      replicas: 2                                  # new: the number of tasks
  web:
    image: ghcr.io/siimrebane/iti8801-notes:a4
    ports:
      - "8089:80"
    deploy:
      replicas: 2
```

`${DB_PASSWORD}`: the swarm does not read `.env`. The value comes from the shell that runs the command: `export DB_PASSWORD=…` first.

The same file, two commands

The file does not know where it will run. The command decides:

	`docker compose up`	`docker stack deploy -c stack.yaml notes`
runs	the Compose tool on one machine	Docker Engine in swarm mode; needs `docker swarm init` done on this machine, otherwise: `this node is not a swarm manager`
`build:`	builds the image	refused: a swarm pulls images from a registry
`deploy:`	reads `replicas`, `resources` and `restart_policy`; ignores placement and update settings	reads all of it
`depends_on:`	starts services in that order; with `condition: service_healthy`, waits	ignored: every service starts at once

Because nothing waits, the api may start before the database accepts connections. It has to retry the connection itself until it works; the Notes api does.

Deploying the stack

```
$ docker stack deploy -c stack.yaml notes
Creating network notes_frontend
Creating network notes_backend
Creating service notes_web
Creating service notes_db
Creating service notes_api
$ docker stack services notes
ID             NAME        MODE         REPLICAS   IMAGE                                  PORTS
w0orennq7xab   notes_api   replicated   2/2        ghcr.io/siimrebane/iti8801-notes:a4
ahuz3gcm4dg2   notes_db    replicated   1/1        postgres:18
44e00pwk7kp6   notes_web   replicated   2/2        ghcr.io/siimrebane/iti8801-notes:a4   *:8089->80/tcp
```

Part	Means
`-c stack.yaml`	the Compose file to read
`notes`	the stack's name; every network and service gets it as a prefix: `notes_api`
`Creating service notes_db`	one service per entry under `services:`, the networks before them
`REPLICAS 2/2`	tasks running of tasks wanted; `notes_db` has no `deploy:` and gets one
`PORTS *:8089->80/tcp`	only `web` publishes a port: 8089 on every node leads to port 80 of a web task

The stack answers

```
$ curl -s -X POST 192.168.42.244:8089/api/notes -H "Content-Type: application/json" -d '{"text":"from swarm"}'
{"id":1,"text":"from swarm","created_at":"2026-09-29T22:10:01.419741Z"}
$ curl -s 192.168.42.244:8089/api/notes
[{"id":1,"text":"from swarm","created_at":"2026-09-29T22:10:01.419741Z"}]
$ docker network ls --filter name=notes
NETWORK ID     NAME              DRIVER    SCOPE
ieutwt7o7t46   notes_backend     overlay   swarm
zb7hrxcmwjlg   notes_frontend    overlay   swarm
```

Part	Means
`192.168.42.244:8089`	the node's address and the published port
the two requests	a note created and read back, as in lecture 5: through a `web` task, an `api` task and the `db` task
`DRIVER overlay`, `SCOPE swarm`	an **overlay** network spans all nodes of the swarm: a task reaches a task on another node by service name, as `api` reaches `db`

What is still missing

The stack runs, and a lost task comes back. Three things it does not handle:

```
$ docker volume ls --filter name=notes
DRIVER    VOLUME NAME
local     notes_pgdata
```

Problem	In this swarm
the database's data	the volume is `local`: it lives on one node. If the `db` task starts on another node, it starts empty
one way in	the tasks run on many nodes, but nothing built in takes port 80 once for every application and sends each request on by host name. You run that program yourself
many people on one swarm	no way to give a team its own part of the swarm, with its own limits and permissions: whoever reaches a manager can change every service

An orchestrator for a whole organisation has to answer these three.

Kubernetes

Kubernetes is the most widely used container orchestrator: open source, started at Google, released in 2014, maintained by the Cloud Native Computing Foundation. It is often written **K8s**: K, eight letters, s.

It comes in **distributions**: packages of the same Kubernetes with different installers and defaults. Every certified distribution accepts the same objects and the same commands.

A cluster

A **cluster** is a set of machines that Kubernetes manages as one.

Part	Is	Does
node	one machine in the cluster, a VM or a physical server	runs containers
control plane	the programs that manage the cluster; they run on one or more nodes	stores the desired state, decides where containers run, reacts to changes

A small cluster runs the control plane on the same nodes as everything else. A production cluster runs it on three separate nodes, so that losing one does not stop the cluster.

The control plane

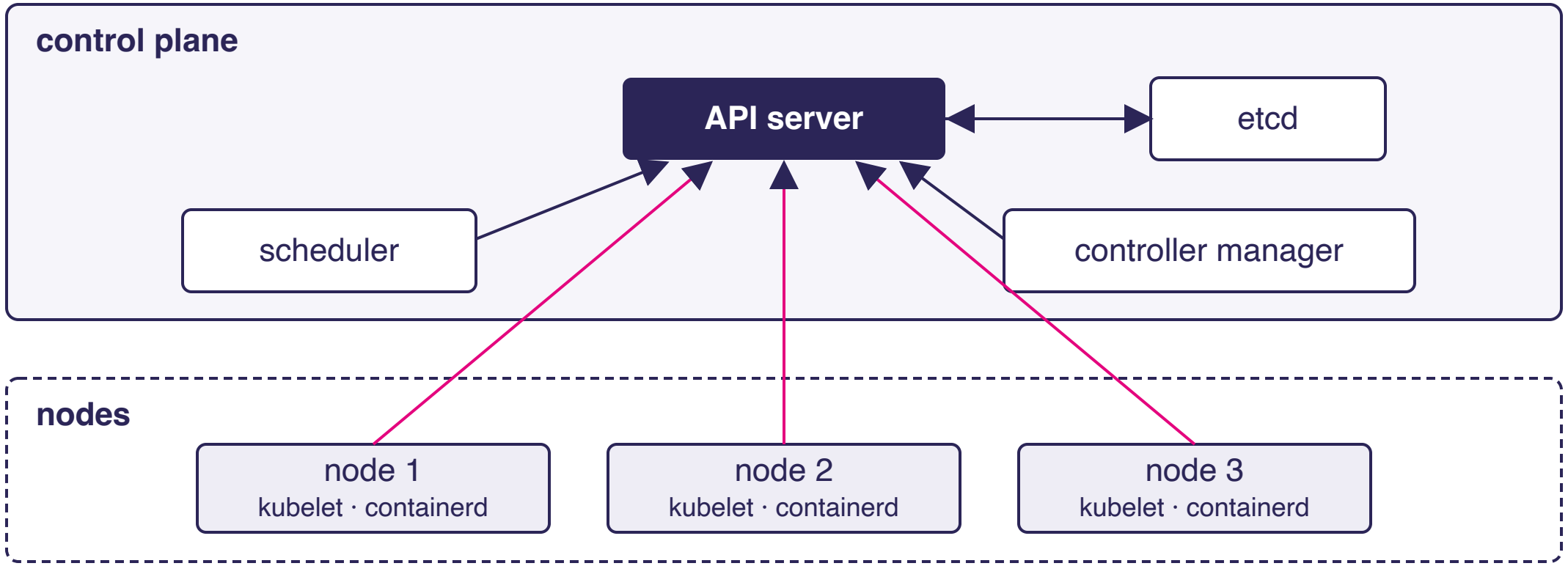
Component	Does
API server	the only entry point: every command, every component, reads and changes the cluster through its HTTPS API
etcd	the database that stores every object: what should exist and what does exist
scheduler	picks a node for each new pod, a group of containers placed together, that has no node yet
controller manager	runs the controllers: loops that compare what should exist with what exists and correct the difference

etcd confirms a change only after a majority of the control-plane nodes have written it to disk. Two things follow. Every change in the cluster waits for those disk writes, so slow disks there slow down everything. And the number of nodes matters: with two, a majority is two, so losing one stops all writes; with three, the remaining two still agree.

A node

Component	Does
kubelet	the agent on every node: asks the API server what should run on this node, starts it, reports its state back
container runtime	starts the containers; k3s uses **containerd**, the same runtime Docker uses underneath
kube-proxy	sets up the node's packet-forwarding rules, so that one stable address reaches the group of containers behind it

The parts together



Every arrow ends at the API server: nothing else talks to etcd, and the components do not talk to each other. Each one watches the API server and acts on what changes.

Distributions

Distribution	Who runs what
AKS, Azure Kubernetes Service	the cloud provider runs the control plane; you pay for the nodes' VMs
EKS, Amazon Elastic Kubernetes Service	the same; the control plane is paid for too
GKE, Google Kubernetes Engine	the same as AKS
k3s	you: one program that installs a whole cluster on your own machines

The same manifests and the same commands work on all of them.

k3s

k3s is a Kubernetes distribution packed into one program of 79 MB. One command installs a node; the first node also runs the control plane.

It is certified Kubernetes, with its own defaults for the parts a cloud would provide:

Part	In k3s	In a cloud cluster
store of the cluster state	one server: SQLite, a database in one file; three servers: etcd, as on the course cluster	etcd, run by the provider
entry point for web traffic	Traefik, a web proxy installed with the cluster	the provider's load balancer
disks for containers	a directory on the node's own disk	the provider's network disks

kubectl and the kubeconfig

`kubectl` is the command-line client: every `kubectl` command is one or more requests to the API server.

A **kubeconfig** file tells `kubectl` where the cluster is and who you are. `kubectl` reads `~/.kube/config`, or the file named in the `KUBECONFIG` environment variable.

Part	Holds
cluster	the API server's address and the certificate of the authority that signed its certificate
user	your credential: a token or a client certificate
context	a named combination of cluster, user and namespace: the group of objects in the cluster you work in
`current-context`	the context `kubectl` uses

Whoever has your kubeconfig acts as you on the cluster: treat it like a password.

A kubeconfig for one student

```yaml
apiVersion: v1
kind: Config
clusters:
- name: iti8801
  cluster:
    server: https://…:6443
    certificate-authority-data: LS0tLS1CRUdJTi…
users:
- name: 000007test
  user:
    token: eyJhbGciOiJSUzI1NiIsImtpZCI6…
contexts:
- name: iti8801
  context: {cluster: iti8801, user: 000007test, namespace: 000007test}
current-context: iti8801
```

```
$ kubectl config get-contexts
CURRENT   NAME      CLUSTER   AUTHINFO     NAMESPACE
*         iti8801   iti8801   000007test   000007test
```

Objects and manifests

Everything in a cluster is an **object**: a record in etcd with a type, a name and fields. A **manifest** is a YAML file that describes objects.

Field	Holds
`apiVersion`	the version of the API group the type belongs to: `v1`, `apps/v1`, `networking.k8s.io/v1`
`kind`	the type: `Pod`, `Deployment`, `Service`, …
`metadata`	the name, the namespace, labels
`spec`	the desired state: what you want
`status`	the observed state: written by the cluster, never by you

`kubectl apply -f file.yaml` sends the manifest to the API server, which creates the objects or updates them to match. Applying the same file again changes nothing.

Controllers

The API server only stores objects. Writing `replicas: 2` into etcd starts nothing; a program has to read it and act. That program is a controller, and controllers are how an orchestrator keeps the result you described true.

A **controller** is a loop that watches one type of object and works to make the observed state match the desired state:

1. read the desired state from the object's `spec`
2. look at what exists
3. create, change or delete things until they match
4. write what it sees to `status`; wait for the next change; start again

Controllers run inside the controller manager, one loop per type of object: a Deployment controller, a ReplicaSet controller, a StatefulSet controller. The scheduler is the same kind of loop as its own program.

Terraform does steps 1–3 once per `apply`. Controllers do them for as long as the cluster runs: deleting something a controller maintains does not remove it, the controller creates it again.

Namespaces

A **namespace** divides one cluster into separate groups of objects. Names must be unique within a namespace, not across the cluster.

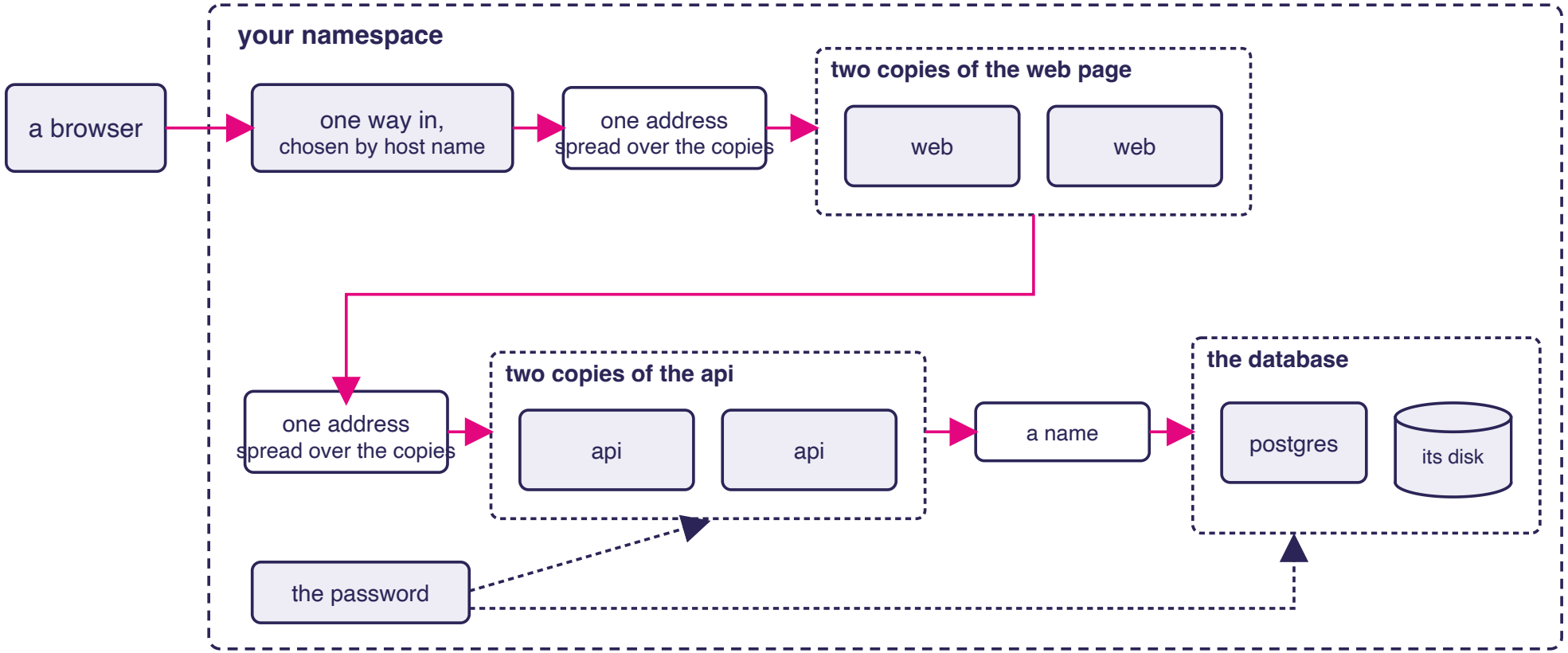
Namespaced: exist inside one namespace	Cluster-wide: belong to no namespace
Pods, Deployments, Services and the other application objects	Nodes, Namespaces

`kubectl` works in one namespace at a time: the context's default, or the one given with `-n <namespace>`.

A new cluster has `default`, and `kube-system` for the cluster's own components.

These namespaces are not the Linux namespaces of lecture 5. A Kubernetes namespace groups objects; it does not isolate processes.

An application on the cluster



This is what has to exist for the Notes application to run; Kubernetes has an object for each part. Solid arrows: the path of a request; each address sends every new connection to one of the copies behind it; dotted: who reads the password.

How it gets there

The application is three YAML files in one directory. A file holds one or more objects, separated by a line `---`:

File	Holds
`db.yaml`	the database's name, and the database with its disk
`api.yaml`	the api: its two copies and their address
`web.yaml`	the web page: its two copies, their address, and the way in

The password is never in a file: one command creates it. Then one command sends every file to the API server, and the controllers do the rest:

```
$ kubectl create secret generic db --from-literal=password="$(openssl rand -hex 16)"
secret/db created
$ kubectl apply -f .
deployment.apps/api created
service/api created
service/db created
statefulset.apps/db created
deployment.apps/web created
service/web created
ingress.networking.k8s.io/notes created
```

One line per object created. To change something: edit the file, `kubectl apply -f .` again. To remove it all: `kubectl delete -f .`.

Pod

Kubernetes does not run containers. It runs **pods**, and a container only exists inside one.

	A container	A pod
is	a process from an image, as in lecture 5	one or more containers that Kubernetes places, names and addresses as one unit
gets	its image, its command, its limits	the node, the IP address, the name, the labels, the `status`
shares	nothing with other containers	with its containers: one network namespace, so one IP and `localhost` between them, and the pod's volumes

In lecture 5's terms: each Docker container got its own network namespace; the containers of a pod share one, so they sit behind one address. Nearly every pod has exactly one container; a second one is a helper that must sit next to the first, such as a proxy.

A pod is disposable: when it is deleted or its node fails, it is gone. Its replacement is a new pod with a new name and a new IP address.

The course cluster

One cluster for the course. Each student has a namespace named after their student code in lower case, and a kubeconfig whose credential is a token bound to a role in that namespace only: the usual application objects there, nothing outside it. The examples use the namespace `000007test`:

```
$ kubectl get pods
No resources found in 000007test namespace.
$ kubectl get nodes
Error from server (Forbidden): nodes is forbidden: User "system:serviceaccount:iti8801-identities:000007test"
  cannot list resource "nodes" in API group "" at the cluster scope
$ kubectl get pods -n kube-system
Error from server (Forbidden): pods is forbidden: User "system:serviceaccount:iti8801-identities:000007test"
  cannot list resource "pods" in API group "" in the namespace "kube-system"
$ kubectl auth can-i create pods
yes
```

`kubectl auth can-i <verb> <type>` asks the API server whether your credential may do something, without doing it.

A pod manifest

Saved as `pod.yaml`:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: hello
  labels:
    app: hello
spec:
  containers:
  - name: web
    image: nginx:1.29-alpine
    ports:
    - containerPort: 80
```

The pod manifest, line by line

Line	Means
`kind: Pod`, `name: hello`	one pod, named `hello`
`labels: app: hello`	a **label**: a key and a value attached to the object, so that commands and other objects can pick it out
`containers:`	the list of containers in the pod; here one, named `web`
`image:`	the image, pulled from its registry as with `docker run`
`containerPort: 80`	records that the container listens on 80; like `EXPOSE`, it opens nothing

Creating it

```
$ kubectl apply -f pod.yaml
pod/hello created
$ kubectl get pods
NAME    READY   STATUS              RESTARTS   AGE
hello   0/1     ContainerCreating   0          0s
$ kubectl get pods -o wide
NAME    READY   STATUS    RESTARTS   AGE   IP           NODE           NOMINATED NODE   READINESS GATES
hello   1/1     Running   0          14s   10.42.5.41   k3s-worker-6   <none>           <none>
```

Column	Means
`READY 1/1`	one of one containers is ready
`STATUS`	`ContainerCreating` while the image is pulled, then `Running`
`RESTARTS`	how many times a container of the pod was restarted
`IP`, `NODE` (with `-o wide`)	the pod's address inside the cluster, and the node it was placed on

What happened: `kubectl describe`

`kubectl describe pod hello` shows the object with its state, and the **events**: what the components did with it.

```
Events:
  Type    Reason     Age   From               Message
  Normal  Scheduled  13s   default-scheduler  Successfully assigned 000007test/hello to k3s-worker-6
  Normal  Pulled     13s   kubelet            spec.containers{web}: Container image "nginx:1.29-alpine" already present on machine
  Normal  Created    13s   kubelet            spec.containers{web}: Container created
  Normal  Started    13s   kubelet            spec.containers{web}: Container started
```

The scheduler chose the node; the kubelet on that node found the image already on its disk, from an earlier pod, and started the container; a new image would show a `Pulling` event first. Events are the first place to look when a pod does not start.

Logs and a command inside

```
$ kubectl logs hello | tail -3
2026/10/05 09:30:20 [notice] 1#1: start worker process 35
2026/10/05 09:30:20 [notice] 1#1: start worker process 36
2026/10/05 09:30:20 [notice] 1#1: start worker process 37

$ kubectl exec hello -- wget -qO- localhost | head -4
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
```

- `kubectl logs <pod>` : the container's standard output and error, as with `docker logs` ; `-f` follows, `--previous` shows the run before the last restart
- `kubectl exec <pod> -- <command>` : runs a command in the pod's container; everything after `--` is the command. `wget -qO- localhost` : fetch the page and print it

Deleting a pod

```
$ kubectl delete pod hello
pod "hello" deleted from 000007test namespace
$ kubectl get pods
No resources found in 000007test namespace.
```

Nothing brought it back: no controller was responsible for this pod. A pod created on its own is used for experiments. Applications run their pods through a controller that keeps the right number of them alive.

When a pod does not run

```
$ kubectl get pods
NAME       READY   STATUS             RESTARTS      AGE
badimage   0/1     ImagePullBackOff   0             80s
crasher    0/1     Error              3 (64s ago)   80s
memhog     0/1     OOMKilled          3 (61s ago)   80s
```

Pod	What it was given	What happened
`badimage`	image `nginx:9.99-doesnotexist`	the pull fails (`ErrImagePull`); the kubelet retries with growing waits: **back-off**, shown as `ImagePullBackOff`
`crasher`	a command that prints `starting` and exits with 1	the kubelet restarts it, with growing waits between restarts; the event is `Back-off restarting failed container`
`memhog`	`tail /dev/zero` with a memory limit of 64 MiB	the kernel kills it at the limit: `OOMKilled` , exit code 137, as in lecture 5

Other Kubernetes versions show the restarting pod as `CrashLoopBackOff` , the name most people know this by.

Finding the cause

```
$ kubectl describe pod badimage | tail -3
  Warning  Failed     36s (x3 over 78s)  kubelet  spec.containers{web}: Error: ErrImagePull
  Normal   BackOff    10s (x4 over 78s)  kubelet  spec.containers{web}: Back-off pulling image "nginx:9.99-doesnotexist"
  Warning  Failed     10s (x4 over 78s)  kubelet  spec.containers{web}: Error: ImagePullBackOff

$ kubectl logs crasher
starting
$ kubectl describe pod crasher | grep -A3 "Last State"
    Last State:     Terminated
      Reason:       Error
      Exit Code:    1
      Started:      Mon, 05 Oct 2026 12:30:51 +0300

$ kubectl describe pod memhog | grep -A3 "Last State"
    Last State:     Terminated
      Reason:       OOMKilled
      Exit Code:    137
      Started:      Mon, 05 Oct 2026 12:30:53 +0300
```

Status	Look at
`ImagePullBackOff`	the events: `ErrImagePull` names the image; check the name, the tag, whether the registry needs a login
restarts climbing	`kubectl logs` and `kubectl logs --previous`; `Last State` for the exit code (`grep -A3`: the line and the 3 after it)
`OOMKilled`	the memory limit against what the program needs

Labels and selectors

A **label** is a key-value pair on an object: `app: api`. Labels carry no meaning for Kubernetes by themselves.

A **selector** is a query on labels: `app=api` matches every object that has that label.

Objects refer to groups of pods by selector, not by name, because pod names change with every replacement:

```
$ kubectl get pods -l app=hello
NAME    READY   STATUS    RESTARTS   AGE
hello   1/1     Running   0          9s
$ kubectl get pods -l app=nothing
No resources found in 000007test namespace.
```

`-l app=hello` passes the selector to `kubectl get`: only the pods with that label.

Deployment

You write a **Deployment**: a pod template, how many copies of it to keep, and a selector that says which pods are its own. The Deployment does not keep the pods itself:

Object	Created by	Keeps
Deployment	you	a pod template and the number of copies; one ReplicaSet per version of the template
ReplicaSet	the Deployment controller	that number of pods alive, created from the template; the pods it owns are those matching the selector
Pod	the ReplicaSet controller	one copy

```
Deployment api
  └─ ReplicaSet api-78c66dbddf          (one per version of the template)
       ├─ Pod api-78c66dbddf-bkxfr
       └─ Pod api-78c66dbddf-qd6kq
```

A ReplicaSet is never written, only seen: the hash in every pod name is its ReplicaSet's, and an update to the template is a new ReplicaSet taking over from the old one.

A Deployment manifest

A Pod manifest gives one pod that nobody replaces. A Deployment wraps the same pod description in `template:` and adds `replicas:`; applications are deployed this way, never as bare pods. `api.yaml`, first object:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api
spec:
  replicas: 2
  selector:
    matchLabels:
      app: api
  template:
    metadata:
      labels:
        app: api
    spec:
      containers:
      - name: notes
        image: ghcr.io/siimrebane/iti8801-notes:a4
        env:
        - name: MODE
          value: api
        ports:
        - containerPort: 8080
```

The Deployment manifest, line by line

Line	Means
`replicas: 2`	keep two pods
`selector: matchLabels: app: api`	the pods that belong to this Deployment
`template:`	the pod to create: the same fields as a Pod manifest
`labels: app: api` in the template	must match the selector, or the Deployment is refused
`env:`	environment variables, as `-e` in `docker run`

The image is the Notes image of lecture 5; `MODE=api` makes it the api. The full manifest also has the database address, health checks and the CPU and memory the container may use.

A pod is deleted: the ReplicaSet replaces it

```
$ kubectl delete pod api-78c66dbddf-bkxfr
pod "api-78c66dbddf-bkxfr" deleted from 000007test namespace
$ kubectl get pods -l app=api
NAME                   READY   STATUS              RESTARTS   AGE
api-78c66dbddf-jrkwm   0/1     ContainerCreating   0          1s
api-78c66dbddf-qd6kq   1/1     Running             0          34s
$ kubectl get pods -l app=api                        # 10 s later
NAME                   READY   STATUS    RESTARTS   AGE
api-78c66dbddf-jrkwm   1/1     Running   0          11s
api-78c66dbddf-qd6kq   1/1     Running   0          44s
```

The ReplicaSet saw one pod where its spec says two, and created one: a new name, a new IP address; ten seconds later both were ready.

A node that stops answering is handled the same way: its pods are marked lost, and their ReplicaSets create replacements on the other nodes.

Scaling

```
$ kubectl scale deployment api --replicas=3
deployment.apps/api scaled
$ kubectl get deployment api
NAME   READY   UP-TO-DATE   AVAILABLE   AGE
api    3/3     3            3           104s
```

Column	Means
`READY 3/3`	ready pods out of desired
`UP-TO-DATE`	pods from the current template
`AVAILABLE`	pods ready for long enough to count as serving

`kubectl scale` changes `spec.replicas` in the cluster, not in your file: the next `kubectl apply -f api.yaml` sets it back to what the file says. Change the file.

The course cluster's nodes

```
$ kubectl get nodes -o wide
NAME          STATUS  ROLES                VERSION       INTERNAL-IP     EXTERNAL-IP
k3s-server-1  Ready   control-plane,etcd   v1.36.4+k3s1  192.168.42.168  193.40.157.249
k3s-server-2  Ready   control-plane,etcd   v1.36.4+k3s1  192.168.42.245  193.40.157.70
k3s-server-3  Ready   control-plane,etcd   v1.36.4+k3s1  192.168.42.150  193.40.157.207
k3s-worker-1  Ready   <none>               v1.36.4+k3s1  192.168.42.96   193.40.157.103
…
k3s-worker-6  Ready   <none>               v1.36.4+k3s1  192.168.42.91   193.40.157.79
```

- three servers run the control plane and etcd; six workers, 8 vCPU and 16 GB each, run the students' pods
- the servers are marked so that only the cluster's own components run on them: every student pod lands on a worker, and a busy namespace cannot slow down the API server or etcd
- `kubectl get nodes` needs the administrator's credential: a student's is refused

Where the pods go

The same three api pods, with their nodes:

```
$ kubectl get pods -l app=api -o wide
NAME                   READY   STATUS    RESTARTS   AGE   IP           NODE
api-78c66dbddf-9cj7m   1/1     Running   0          9s    10.42.7.52   k3s-worker-1
api-78c66dbddf-jrkwm   1/1     Running   0          20s   10.42.3.18   k3s-worker-4
api-78c66dbddf-qd6kq   1/1     Running   0          53s   10.42.5.44   k3s-worker-6
```

The scheduler placed each pod of the ReplicaSet on a different worker: by default it spreads a ReplicaSet's pods over nodes, so that losing one node loses as few of them as possible. The pod's address, `10.42.x.y`, is in the cluster's own network: each worker owns one `10.42.n.0/24` of it.

A node taken out of service

`kubectl drain` takes a node out for maintenance: it marks the node unschedulable (**cordon**) and evicts its pods, whose controllers then create replacements on other nodes. An administrator's command, run while a Deployment `boardgames` had four pods, one per worker:

```
$ kubectl drain k3s-worker-6 --ignore-daemonsets --delete-emptydir-data
node/k3s-worker-6 cordoned
evicting pod kube-system/traefik-685fd4785d-jjzgb
evicting pod 000007test/boardgames-7dfc45868f-tjjcf
…
node/k3s-worker-6 drained
$ kubectl get pods -o wide            # as the student, 15 s later
boardgames-7dfc45868f-9lbbl  1/1  Running  19s  k3s-worker-3
boardgames-7dfc45868f-wmxjg  1/1  Running  37m  k3s-worker-4
boardgames-7dfc45868f-xtqcj  1/1  Running  29s  k3s-worker-2
boardgames-7dfc45868f-xwsxk  1/1  Running  29s  k3s-worker-1
```

- the output also lists a pod from `kube-system`: the cluster's own components are evicted and replaced the same way. The boardgames API, requested every 0.5 s during the drain: 100 of 100 answers `200`
- `kubectl uncordon k3s-worker-6` lets pods onto the node again
- `db-0` cannot move like that: its volume is a directory on `k3s-worker-1`, so a drain of that node leaves the database waiting until the node is back

A rolling update

Any change to the pod template, an image tag or an environment variable, starts a **rolling update**: a new ReplicaSet grows while the old one shrinks, a pod at a time, and a new pod counts only once it is ready. `web` is a second Deployment, the Notes frontend, written like `api` with `MODE=frontend`:

```
$ kubectl set env deployment/web LOG_LEVEL=debug
deployment.apps/web env updated
$ kubectl rollout status deployment/web
Waiting for deployment "web" rollout to finish: 0 out of 2 new replicas have been updated...
Waiting for deployment "web" rollout to finish: 1 out of 2 new replicas have been updated...
Waiting for deployment "web" rollout to finish: 1 old replicas are pending termination...
deployment "web" successfully rolled out
$ kubectl get replicasets -l app=web
NAME            DESIRED   CURRENT   READY   AGE
web-d64b6569c   2         2         2       8s
web-d67d8b8c4   0         0         0       65s
```

- `kubectl set env deployment/web LOG_LEVEL=debug`: change one environment variable of the template, as editing the file and applying it would
- `kubectl rollout status`: wait for the rollout and report its steps
- the old ReplicaSet stays with zero pods: it is the way back

A rollout that cannot finish

The image tag changed to one that does not exist. `kubectl set image deployment/web notes=…`: the image of the container named `notes` in the template:

```
$ kubectl set image deployment/web notes=ghcr.io/siimrebane/iti8801-notes:does-not-exist
deployment.apps/web image updated
$ kubectl get pods -l app=web
NAME                   READY   STATUS         RESTARTS   AGE
web-6b8f969b67-d49lm   0/1     ErrImagePull   0          41s
web-d64b6569c-bc7lt    1/1     Running        0          43s
web-d64b6569c-jrjbf    1/1     Running        0          49s
$ kubectl get deployment web
NAME   READY   UP-TO-DATE   AVAILABLE   AGE
web    2/2     1            2           106s
```

The new pod never became ready, so no old pod was removed: both old pods kept serving. Requests to the application during the rollout and the undo answered `200` every time.

Rolling back

```
$ kubectl rollout history deployment/web
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
3         <none>
$ kubectl rollout undo deployment/web
Warning: resource deployments/web was previously managed with 'kubectl apply'. Rolling back will not
update the kubectl.kubernetes.io/last-applied-configuration annotation, …
deployment.apps/web rolled back
$ kubectl get pods -l app=web
NAME                   READY   STATUS    RESTARTS   AGE
web-d64b6569c-bc7lt    1/1     Running   0          53s
web-d64b6569c-jrjbf    1/1     Running   0          59s
```

- `rollout undo` switches back to the previous ReplicaSet's template
- the warning: the cluster now differs from the file that was last applied. The file is the record; after an undo, fix the file and apply it again

Secrets

A **Secret** holds passwords, tokens and keys as key-value pairs, and a container gets them as environment variables or as files. It is its own kind of object so that reading it can be allowed separately from reading everything else.

```
        env:
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: db
              key: password
        - name: DATABASE_URL
          value: postgres://postgres:$(DB_PASSWORD)@db:5432/notes?sslmode=disable
```

`secretKeyRef`: the value of the key `password` in the Secret named `db`. `$(DB_PASSWORD)` in a later variable is replaced with the value of an earlier one in the same list.

A Secret is not encrypted

```
$ kubectl create secret generic db --from-literal=password="$(openssl rand -hex 16)"
secret/db created
$ kubectl get secret db -o yaml | grep -A1 "^data:"
data:
  password: OGRkM2M2MDczNzQ2ZDI2MDI3YWE5MmI3ZTc5ZGJmOGI=
$ kubectl get secret db -o jsonpath="{.data.password}" | base64 -d
8dd3c6073746d26027aa92b7e79dbf8b
```

- `create secret generic db --from-literal=password=…` : a Secret named `db` with one key; `openssl rand -hex 16` makes a random 32-character password
- `-o yaml` : the object as the cluster stores it; `-o jsonpath="{.data.password}"` : one field of it, instead of a table
- the value is stored **base64-encoded**: a way of writing bytes as text, reversed by anyone with `base64 -d`
- anyone allowed to read Secrets in the namespace reads the password. Encryption of Secrets inside etcd is a cluster setting
- a Secret manifest in git is the password in git: create Secrets with `kubectl create secret`, not from a committed file

Requests and limits

Each container can declare:

Field	Means	Enforced by
`requests.cpu`, `requests.memory`	the amount reserved for it	the scheduler: a pod goes only to a node with that much unreserved
`limits.cpu`	the most CPU it may use	the kernel's cgroups: slowed down above it
`limits.memory`	the most memory it may use	the kernel's cgroups: killed above it, `OOMKilled`

CPU is counted in cores: `500m` is 500 millicores, half a core.

```
resources:
  requests: {cpu: 50m, memory: 64Mi}
  limits: {cpu: 200m, memory: 128Mi}
```

A pod that fits no node

A pod requesting 64 GiB of memory, created by the administrator in a scratch namespace without a quota:

```
$ kubectl -n tmp-pending get pod huge
NAME   READY   STATUS    RESTARTS   AGE
huge   0/1     Pending   0          8s
$ kubectl -n tmp-pending describe pod huge | tail -1
  Warning  FailedScheduling  8s  default-scheduler  0/9 nodes are available: 3 node(s) had untolerated taint(s), 6 Insufficient memory. …
$ kubectl describe node k3s-worker-1 | grep -A6 "^Allocatable"
Allocatable:
  cpu:                8
  ephemeral-storage:  97769715226
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  memory:             16370392Ki
  pods:               110
```

`Pending` means no node was chosen, and the event says why for each node: the three servers are reserved for the cluster's own components, the six workers have 16 GiB each. It stays `Pending` until a node with room appears. **Allocatable** is what a node offers to pods, and `pods: 110` is the default limit on pods per node; `kubectl describe node` is an administrator's command.

Your namespace's quota

A **ResourceQuota** caps the total a namespace may reserve. A **LimitRange** gives containers default requests and limits when their manifest has none.

```
$ kubectl describe quota
Resource                Used   Hard
limits.cpu              1300m  4
limits.memory           1Gi    4Gi
persistentvolumeclaims  1      4
pods                    5      20
requests.cpu            300m   500m
requests.memory         384Mi  2Gi
requests.storage        1Gi    4Gi
services.loadbalancers  0      0
services.nodeports      0      0
```

A pod that would go over the quota is refused when it is created:

```
$ kubectl apply -f big.yaml
Error from server (Forbidden): error when creating "big.yaml": pods "big" is forbidden: exceeded quota: student-quota,
  requested: requests.memory=2500Mi, used: requests.memory=320Mi, limited: requests.memory=2Gi
```

A Deployment the quota stops

The quota counts **limits** too. Eight pods with `limits.cpu: 500m` each, plus the database's 500m, need 4.5 CPU of limits; the namespace may have 4:

```
$ kubectl scale deployment boardgames --replicas=8
$ kubectl get deployment boardgames
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
boardgames   7/8     7            7           35m
$ kubectl describe replicaset -l app=boardgames | grep FailedCreate | tail -1
  Warning  FailedCreate  39s (x32 over 10m)  replicaset-controller  … Error creating: pods "boardgames-7dfc45868f-dk22s"
  is forbidden: exceeded quota: student-quota, requested: limits.cpu=500m, used: limits.cpu=4, limited: limits.cpu=4
```

`kubectl get pods` shows only the pods that exist and no error: the refusal is in the **ReplicaSet's events**, because the ReplicaSet is the one that tried to create the pod.

Reserved and used

`kubectl top` shows what the pods use now, measured by the metrics server that k3s installs:

```
$ kubectl top pods
NAME                   CPU(cores)   MEMORY(bytes)
api-78c66dbddf-jrkwm   2m           3Mi
api-78c66dbddf-qd6kq   2m           4Mi
db-0                   12m          35Mi
web-d64b6569c-bc7lt    2m           2Mi
web-d64b6569c-jrjbf    2m           3Mi
```

Five pods use about 47 MiB; their requests add up to 384 MiB. Requests are what the scheduler counts and what the quota caps; set them near real use, with room for peaks.

Probes

A **probe** is a check the kubelet runs against a container, as an HTTP request, a TCP connection or a command. Three probes, three questions:

Probe	Asks	When it fails	For
readiness	can this pod take requests right now?	the pod keeps running but gets no traffic until it passes again; a rollout waits for new pods to pass	conditions that pass by themselves: starting up, the database away for a moment, overload
liveness	is the process still working, or stuck?	the kubelet restarts the container	a state the program cannot leave on its own: a deadlock, a hung loop
startup	has the program finished starting?	the other two do not run until it has passed once	programs that start slowly, so liveness does not kill them while they start

```
readinessProbe:
  httpGet: {path: /health, port: 8080}
  periodSeconds: 5
livenessProbe:
  httpGet: {path: /health, port: 8080}
  periodSeconds: 10
```

`httpGet`: an HTTP request to this path and port of the container; `periodSeconds`: how often. `/health` of the Notes api answers `200` only when the database answers, so here liveness depends on the database too: a long database outage would restart the api pods. A liveness probe should test the process itself; readiness is the one that may look at dependencies.

A readiness probe that fails

The api's probe path changed to `/healthz`, which the Notes api does not have:

```
$ kubectl get pods -l app=api
NAME                   READY   STATUS    RESTARTS   AGE
api-775566b7-csj7s     0/1     Running   0          35s
api-78c66dbddf-jrkwm   1/1     Running   0          2m
api-78c66dbddf-qd6kq   1/1     Running   0          2m33s
$ kubectl describe pod api-775566b7-csj7s | tail -1
  Warning  Unhealthy  4s (x7 over 34s)  kubelet  spec.containers{notes}: Readiness probe failed: HTTP probe failed with statuscode: 404
$ kubectl rollout status deployment/api --timeout=20s
Waiting for deployment "api" rollout to finish: 1 out of 2 new replicas have been updated...
error: timed out waiting for the condition
```

The new pod runs but never becomes ready: it gets no traffic, and the rollout stops with the two old pods serving.

Reaching the api

The frontend has to connect to the api. Where?

```
$ kubectl get pods -l app=api -o wide
NAME                   READY   STATUS    RESTARTS   AGE     IP            NODE
api-78c66dbddf-jrkwm   1/1     Running   0          2m36s   10.42.3.18    k3s-worker-4
api-78c66dbddf-qd6kq   1/1     Running   0          3m9s    10.42.5.44    k3s-worker-6
```

- two addresses today; after the next replacement or rollout, different ones: a replaced pod is a new pod with a new address
- nothing in the cluster answers the name `api` yet. In Compose the service name did that for free
- the frontend would have to list the pods itself, follow every change, and pick one

A group of pods needs one name and one address that stay, whatever happens to the pods behind them.

Service

Pods come and go with new addresses, and there are several of them. A **Service** gives a group of pods one stable name and one stable virtual address, and spreads connections over the pods that are ready. `api.yaml`, second object, after the `---` line:

```
apiVersion: v1
kind: Service
metadata:
  name: api
spec:
  selector:
    app: api
  ports:
  - port: 8080
```

- `selector: app: api`: the pods behind the Service, found by label. A Service and a Deployment never refer to each other: both select pods by label, and the labels on the pod template are the only thing that joins them
- `port: 8080`: the port the Service answers on; without `targetPort`, the pods' port is the same
- the Service's address is a **ClusterIP**: reachable only inside the cluster. kube-proxy's rules on every node send connections to it on to one of the pods

The Services of the Notes application

The three Services of the Notes application, one from each file:

```
$ kubectl get services
NAME   TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
api    ClusterIP   10.43.76.136   <none>        8080/TCP   3m9s
db     ClusterIP   None           <none>        5432/TCP   3m9s
web    ClusterIP   10.43.86.137   <none>        80/TCP     3m8s
```

Column	Means
`CLUSTER-IP`	the Service's virtual address, chosen from the cluster's Service range when it is created
`None` for `db`	a Service without a virtual address
`EXTERNAL-IP` `<none>`	none of them is reachable from outside the cluster

Names: the cluster's DNS

The cluster runs a DNS server, CoreDNS, and every pod uses it. Each Service gets a name:

Name	Works from
`api`	pods in the same namespace
`api.000007test`	pods in any namespace
`api.000007test.svc.cluster.local`	the full name

The name works only inside the cluster: your own machine does not use the cluster's DNS. To ask from inside, a throwaway pod that runs one command and is deleted:

```
$ kubectl run dns --rm -i --restart=Never --image=alpine:3.22 -- nslookup api
Name:   api.000007test.svc.cluster.local
Address: 10.43.76.136
```

`kubectl run dns --image=alpine:3.22` : one pod named `dns` from an image that has `nslookup` ; `--restart=Never` : a pod that runs once, not a Deployment; `-i` : show its output; `--rm` : delete it afterwards. Everything after `--` is its command.

The two Services you write

The cluster is a network of its own, like the VPC of weeks 2 to 4: pods have addresses in `10.42.0.0/16`, Services in `10.43.0.0/16`, and nothing outside the nodes can route to either. The nodes' public addresses are all the internet sees.

Type	The name resolves to	Use it for
`ClusterIP` (the default)	one virtual address, the `CLUSTER-IP`; kube-proxy sends each connection on to a ready pod	anything other pods call, and anything the ingress controller, itself a pod, forwards to: `api`, `web`
headless: `clusterIP: None`	the pod's own address: no virtual address in between	one specific pod, not "any ready copy": the database

```
$ kubectl run dns --rm -i --restart=Never --image=alpine:3.22 -- nslookup db
Name:   db.000007test.svc.cluster.local
Address: 10.42.5.47
```

`api` resolved to `10.43.76.136`, a Service address; `db` resolves to `10.42.5.47`, the database pod itself. Crossing the line into the cluster is the ingress controller's job, as the load balancer in the public subnet was in week 4. Two more types, `NodePort` and `LoadBalancer`, give one Service an address outside the cluster without an Ingress; the course quota allows neither (`services.nodeports 0`, `services.loadbalancers 0`), and nothing here needs them.

Ingress

A cluster runs many web applications and has few public addresses. An **Ingress** is a rule: requests for this host name and path go to this Service.

The rules are carried out by an **ingress controller**: a web proxy running in the cluster that reads every Ingress and routes by them. k3s installs **Traefik** as its ingress controller, reached through a `LoadBalancer` Service on ports 80 and 443 of the nodes.

```
internet → node:80 → Traefik → (host 000007test.k8s.digivader.com) → Service web → web pods
                             → (host 251234iaib.k8s.digivader.com) → another student's Service
```

In the course cluster one ingress controller serves every student's Ingress, and the host name tells them apart: the DNS record `*.k8s.digivader.com` points every such name at the cluster's nodes, so your host name is `<your code in lower case>.k8s.digivader.com` and works in any browser. Plain HTTP this week: HTTPS is the ingress controller's job too, a certificate per host name that it can obtain itself, and it is not set up on the course cluster.

An Ingress manifest

`web.yaml`, third object, after the web Deployment and its Service:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: notes
spec:
  rules:
  - host: 000007test.k8s.digivader.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: web
            port:
              number: 80
```

```
$ kubectl get ingress
NAME    CLASS     HOSTS                          ADDRESS                                PORTS   AGE
notes   traefik   000007test.k8s.digivader.com   193.40.157.103,193.40.157.158,…   80      3m14s
```

`pathType: Prefix` with `/`: every path under the host. `CLASS traefik`: the ingress controller that carries the rule out; `ADDRESS`: the nodes it listens on, all nine.

Through the Ingress, from outside

The DNS record `*.k8s.digivader.com` points every such name at the cluster's nodes, so the host name works from anywhere:

```
$ curl -s -X POST http://000007test.k8s.digivader.com/api/notes -H "Content-Type: application/json" -d '{"text":"from kubernetes"}'
{"id":1,"text":"from kubernetes","created_at":"2026-10-05T09:35:14.552613Z"}

$ curl -s http://000007test.k8s.digivader.com/api/notes
[{"id":1,"text":"from kubernetes","created_at":"2026-10-05T09:35:14.552613Z"}]

$ curl -s -o /dev/null -w "%{http_code}\n" http://193.40.157.249/          # a node's address, no host name
404
```

`-o /dev/null -w "%{http_code}\n"` : print only the status code. The request went: Traefik → Service `web` → a web pod → Service `api` → an api pod → `db` → the database pod. Without a host name that matches a rule, Traefik answers 404.

Running the database: why a Deployment cannot

A container's files disappear with the container. Postgres needs one directory that is its own and stays: a **PersistentVolumeClaim** asks the cluster for storage of a given size, and on this cluster the volume behind it is a directory on the disk of one worker.

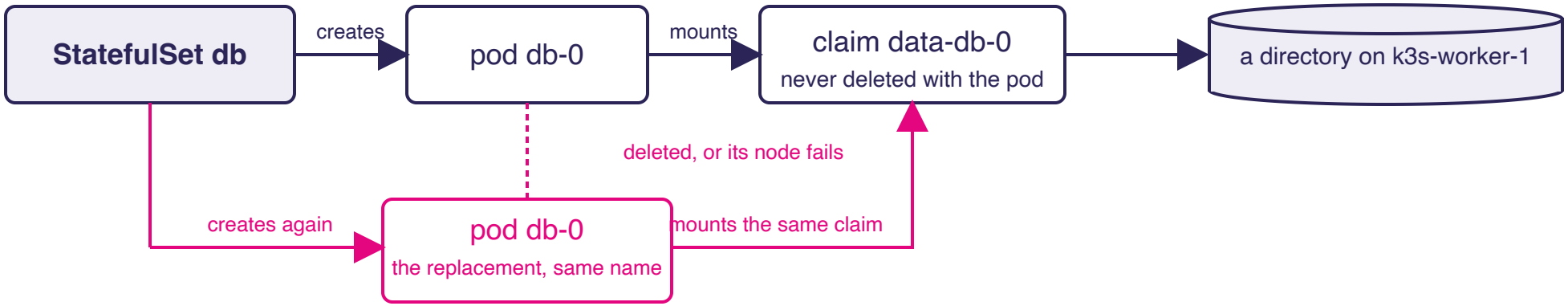
A Deployment is built for interchangeable copies, and a database is not one:

With a Deployment	What goes wrong
the pod template mounts a claim	every replica mounts the same claim: two Postgres copies on one directory
a pod is replaced	the new pod has a new name and no way to say "that disk was mine"
the pod lands on another worker	the directory is on the old one; a claim here can be mounted by one node at a time (`ReadWriteOnce`)

The database needs a pod with a stable identity, and a claim that belongs to that identity, not to the pod.

StatefulSet

A **StatefulSet** runs pods with a stable name, `db-0`, and makes one claim per pod from a template, named after the pod:



The claim stays when the pod goes. The replacement is again `db-0`, so it mounts `data-db-0` again, and the volume's record of where its directory is sends the pod back to the same worker. Only deleting the claim deletes the data. A second pod would be `db-1` with `data-db-1`, started after `db-0` is ready: the numbering gives an order, for programs that need one.

The database as a StatefulSet

`db.yaml`, second object; the headless Service `db` is the first:

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: db
spec:
  serviceName: db
  replicas: 1
  selector:
    matchLabels: {app: db}
  template:
    metadata:
      labels: {app: db}
    spec:
      containers:
      - name: postgres
        image: postgres:18-alpine
        env:
        - name: POSTGRES_DB
          value: notes
        - name: POSTGRES_PASSWORD
          valueFrom:
            secretKeyRef: {name: db, key: password}
        volumeMounts:
        - name: data
          mountPath: /var/lib/postgresql
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 1Gi
```

The StatefulSet, line by line

Line	Means
`serviceName: db`	the headless Service that gives the pods their DNS names
`valueFrom: secretKeyRef`	the password from the Secret `db`, key `password`
`volumeMounts: data` at `/var/lib/postgresql`	where Postgres 18 keeps its data
`volumeClaimTemplates: data`	for each pod, a claim named `data-<pod>`: `data-db-0`
`storage: 1Gi`	its size; counted in the namespace's `requests.storage`

The full file also has a readiness probe that runs `pg_isready`, and resources.

The data outlives the pod

```
$ kubectl get pvc
NAME        STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS
data-db-0   Bound    pvc-8e7553e6-826e-42cf-844e-c151b269ef58   1Gi        RWO            local-path

$ kubectl exec db-0 -- psql -U postgres -d notes -c "select count(*) from notes"
 count
-------
     1
$ kubectl delete pod db-0
pod "db-0" deleted from 000007test namespace
$ kubectl get pods db-0
NAME   READY   STATUS    RESTARTS   AGE
db-0   1/1     Running   0          9s
$ kubectl exec db-0 -- psql -U postgres -d notes -c "select count(*) from notes"
 count
-------
     1
```

`psql` is Postgres's command-line client; `-c` runs one query. The new `db-0` mounted the same claim: the note is still there. Deleting the claim deletes the data.

Traffic between namespaces

By default every pod can connect to every other pod in the cluster, in any namespace. A **NetworkPolicy** limits which connections reach the pods it selects.

Each namespace in the course cluster has this policy: connections to its pods are allowed from pods in the same namespace and from `kube-system`, where the ingress controller runs; everything else is blocked.

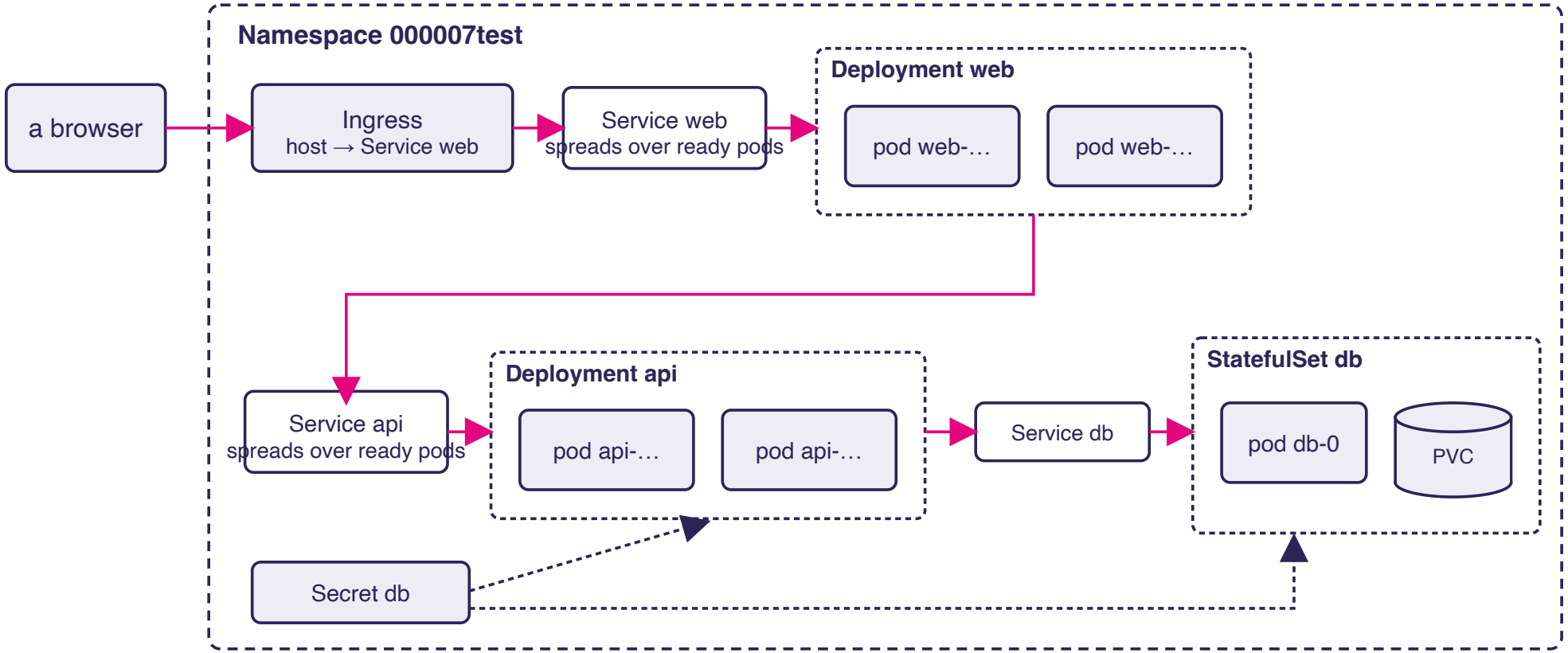
```
# from a pod in another namespace, "other":
$ kubectl -n other run probe --rm -i --restart=Never --image=alpine:3.22 -- sh -c "sleep 5; wget -qO- -T 5 http://api.000007test:8080/health"
wget: can't connect to remote host (10.43.76.136): Connection refused
# from a pod in the same namespace:
$ kubectl run probe --rm -i --restart=Never --image=alpine:3.22 -- sh -c "sleep 5; wget -qO- -T 5 http://api:8080/health"
{"db":"ok","mode":"api","status":"ok"}
```

`sleep 5` first: the rules for a new pod take a moment to be in place; `wget -T 5`: give up after 5 seconds.

From Compose to Kubernetes

In `compose.yaml`	In Kubernetes
a service with `image:`	a Deployment (a StatefulSet for a database)
the service name others connect to	a Service with that name
`ports: "80:80"`	an Ingress, reached through the ingress controller
`environment:`	`env:` in the pod template
`.env` with the password	a Secret, read with `valueFrom: secretKeyRef`
a named volume	a PersistentVolumeClaim
`healthcheck:`, `depends_on: condition: service_healthy`	readiness probes; a program that retries its connections
`networks:` that separate services	NetworkPolicies
`restart: unless-stopped`	built in: controllers always replace

The Notes application: the objects



A request: Traefik → Service `web` → a web pod → Service `api` → an api pod → `db` → the database pod. `db` is the headless Service; the Secret is read by the api pods and the database pod; the claim `data-db-0` is the database's disk.

The Notes application on the cluster

```
$ kubectl get all
NAME                          READY   STATUS    RESTARTS   AGE
pod/api-78c66dbddf-jrkwm      1/1     Running   0          3m7s
pod/api-78c66dbddf-qd6kq      1/1     Running   0          3m40s
pod/db-0                      1/1     Running   0          24s
pod/web-d64b6569c-bc7lt       1/1     Running   0          2m36s
pod/web-d64b6569c-jrjbf       1/1     Running   0          2m42s

NAME          TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
service/api   ClusterIP   10.43.76.136   <none>        8080/TCP   3m40s
service/db    ClusterIP   None           <none>        5432/TCP   3m40s
service/web   ClusterIP   10.43.86.137   <none>        80/TCP     3m39s

NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/api   2/2     2            2           3m40s
deployment.apps/web   2/2     2            2           3m39s

NAME                                DESIRED   CURRENT   READY   AGE
replicaset.apps/api-775566b7        0         0         0       102s
replicaset.apps/api-78c66dbddf      2         2         2       3m40s
replicaset.apps/web-6b8f969b67      0         0         0       2m34s
replicaset.apps/web-d64b6569c       2         2         2       2m42s
replicaset.apps/web-d67d8b8c4       0         0         0       3m39s

NAME                  READY   AGE
statefulset.apps/db   1/1     3m39s
```

Three files, `db.yaml`, `api.yaml`, `web.yaml`, applied with `kubectl apply -f .`; the Secret created before them. The ReplicaSets at 0 are the rollouts' history: the failed image, the probe that never passed, the first version of web.

Kubernetes run by a cloud

With AKS, EKS or GKE, the provider runs the control plane: the API server, etcd, their upgrades and their availability. You choose the Kubernetes version and **node pools**: groups of VMs of one size that become the nodes.

The same manifests work. What the cloud adds:

Object	In k3s on your own VMs	In a cloud cluster
Service `type: LoadBalancer`	a forwarder on the nodes' own addresses	a new cloud load balancer with its own public address
PersistentVolumeClaim	a directory on one node's disk	a cloud disk, attached to whichever node in the same zone the pod moves to
identity of a pod	none outside the cluster	a cloud identity, so the pod reaches the provider's services without a stored password
number of nodes	fixed	node pools that can grow and shrink with the pods' requests

Price: the nodes are billed as VMs. The control plane: EKS $0.10 per cluster per hour, while the Kubernetes version is in standard support; AKS's Free tier nothing, without an availability guarantee.

Managed container services

A **managed container service** runs containers from an image without a cluster you can see: you give the image, CPU and memory, the number of copies or a scaling rule, and the port; the provider does the rest.

Service	Provider	Is
Azure Container Apps	Azure	apps from images, scaled by requests or events, down to zero; runs on Kubernetes that you do not see
Amazon ECS with Fargate	AWS	AWS's own orchestrator (ECS); Fargate runs each task without VMs you manage
Google Cloud Run	Google	containers that answer HTTP requests or run jobs, scaled with the requests, down to zero
Azure Container Instances	Azure	single containers or small groups, started on demand, no scaling

Each one has its own configuration format, command-line tool and limits. Moving an application from one to another means writing its deployment again; the image stays the same.

Choosing where containers run

Question	Points towards
Who runs the platform, and how much of their time can it take?	little time: a managed container service; people for it: Kubernetes
How many services, and how different from each other?	one or two: a managed service or one VM; many, with their own scaling, storage, rules: Kubernetes
Must it move between clouds or run on own hardware?	Kubernetes: the same manifests on AKS, EKS, GKE, k3s
Is it idle most of the time?	services that scale to zero: Container Apps, Cloud Run
How much control over networking, storage and placement is needed?	the more the provider runs, the less of it

The container image is the same in all of them: an image from `docker build` runs on each.

The commands for objects

Command	Does
`kubectl apply -f file.yaml`, `-f .`	create or update the objects in a file, or in every file of a directory
`kubectl get pods`, `… -o wide`, `… -l app=api`	list; with node and IP; only those with a label
`kubectl get all`	the common application objects of the namespace
`kubectl describe <type> <name>`	one object with its state and events
`kubectl delete -f file.yaml`, `kubectl delete pod <name>`	delete what a file describes; one object
`kubectl get <type> <name> -o yaml`	the object as the cluster stores it, with `status`
`kubectl explain deployment.spec`	the documentation of a field, from the cluster itself

The commands for running things

Command	Does
`kubectl logs <pod>`, `-f`, `--previous`	output; followed; of the run before the last restart
`kubectl exec <pod> -- <command>`, `-it … -- sh`	a command in the container; an interactive shell, if the image has one
`kubectl rollout status / history / undo deployment/<name>`	follow a rollout; list revisions; go back one
`kubectl scale deployment <name> --replicas=3`	change the number of pods, until the next `apply`
`kubectl port-forward service/web 8080:80`	connect port 8080 on your machine to the Service, through the API server
`kubectl top pods`	CPU and memory in use now
`kubectl auth can-i <verb> <type>`	whether your credential may do it



Questions