ITI8801 · Cloud Architectures and DevOps

Kubernetes: operating an application

Siim Rebane · TalTech · Autumn 2026

What we built

- One namespace, two environments in it: **dev** (every push to `main`) and **staging** (every tag)
- Two manifest files, four placeholders `$ENV $HOST $IMAGE $REPLICAS`, filled by `envsubst` in the deploy job, applied with your kubeconfig

```
$ kubectl get deployments,services,ingresses,statefulsets
NAME                             READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/api-dev          1/1     1            1           18s
deployment.apps/api-staging      2/2     2            2           17s

NAME                     TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
service/api-dev          ClusterIP   10.43.216.6     <none>        80/TCP     18s
service/api-staging      ClusterIP   10.43.158.68    <none>        80/TCP     17s
service/db-dev           ClusterIP   None            <none>        5432/TCP   12d
service/db-staging       ClusterIP   None            <none>        5432/TCP   12d

NAME                                    CLASS     HOSTS                                      PORTS   AGE
ingress.networking.k8s.io/api-dev       traefik   000007test-dev.k8s.digivader.com           80      18s
ingress.networking.k8s.io/api-staging   traefik   000007test-staging.k8s.digivader.com       80      17s

NAME                             READY   AGE
statefulset.apps/db-dev          1/1     12d
statefulset.apps/db-staging      1/1     12d
```

- It runs. This lecture is about what it takes to **operate** it: change a setting, follow the load, survive maintenance, and ship all of that per environment with a way back.

A setting

- Where the configuration is today:
  - CI/CD variables → the four placeholders: environment name, host, image, replica count
  - the Secret → the database password
  - the Deployment's pod template → `DATABASE_URL`, put together from the two above
- The new version of the service reads one more value, `ENVIRONMENT`, and shows it at `/info`:

```
$ curl -s http://000007test-staging.k8s.digivader.com/info
{"environment":""}
```

- Where does `ENVIRONMENT=staging` go?
  - **in the image** → a different value needs a different image; dev and staging no longer run the same image
  - **as a fifth placeholder** → one more CI variable; every change = a commit, a pipeline, a rollout of a file both environments share
  - **in the Secret** → it is not a secret; access to Secrets is kept narrow
- What is missing: an object that holds the settings of one environment, by themselves

ConfigMap

- A **ConfigMap** is a Kubernetes object like a Deployment or a Secret: it has a name, it lives in a namespace, it is written as a manifest and applied
- It holds settings as keys and values; the values are text
- By itself it is attached to nothing. A pod refers to it by name, and the pod decides how to read it.
- It can be changed without touching the image or the Deployment
- It is for ordinary settings; passwords go in a Secret

```
$ kubectl create configmap api-staging --from-literal=ENVIRONMENT=staging --from-literal=LOG_LEVEL=info
configmap/api-staging created
$ kubectl get configmap api-staging -o yaml
apiVersion: v1
data:
  ENVIRONMENT: staging
  LOG_LEVEL: info
kind: ConfigMap
metadata:
  name: api-staging
  namespace: 000007test
```

- Two keys. `kubectl create configmap` typed the manifest for us.

The Deployment reads it

- A ConfigMap does nothing until a pod refers to it. The reference goes in the pod template, so it is a change to the Deployment manifest, `k8s/api.yaml`, once for both environments.
- Three lines in the container's spec:

```
        envFrom:
        - configMapRef: {name: api-$ENV}
```

- `envFrom`: every key of the ConfigMap becomes an environment variable with that name
- The placeholder picks `api-dev` or `api-staging`, like everywhere else in the file

```
$ envsubst '$ENV $HOST $IMAGE $REPLICAS' < k8s/api.yaml | kubectl apply -f -
deployment.apps/api-staging configured
$ kubectl rollout status deployment/api-staging
deployment "api-staging" successfully rolled out
$ curl -s http://000007test-staging.k8s.digivader.com/info
{"environment":"staging"}
```

- A changed pod template is a rollout: new pods, with the variables; the old pods go

The pod reads it as files

- The same ConfigMap, the other way into the pod: the kubelet writes each key as a file into the container, at the path the template names. Nothing exists on any disk; the object is still the only thing.

```
        volumeMounts:
        - name: config
          mountPath: /config
      volumes:
      - name: config
        configMap:
          name: api-$ENV
```

```
$ kubectl exec deployment/api-staging -- env | grep -E 'ENVIRONMENT|LOG_LEVEL'
LOG_LEVEL=info
ENVIRONMENT=staging
$ kubectl exec deployment/api-staging -- ls /config
ENVIRONMENT
LOG_LEVEL
$ kubectl exec deployment/api-staging -- cat /config/LOG_LEVEL
info
```

- Variables suit single values a program reads at start. Files suit whole configuration files, an `nginx.conf` or a `settings.json`: one key, the file's content as its value.
- One variable from one key is the third form: `valueFrom: configMapKeyRef: {name: api-staging, key: LOG_LEVEL}` under `env:`
- The program does not know where a value came from. The image is the same in dev and in staging.

A change

- `LOG_LEVEL` goes from `info` to `debug`. That is a change to the ConfigMap object only: no new image, no change to the Deployment, no deploy.
- `kubectl patch`: change one field of an object that exists, in place; the same as editing its manifest and applying it again

```
$ kubectl patch configmap api-staging --type merge -p '{"data":{"LOG_LEVEL":"debug"}}'
configmap/api-staging patched
```

- Do the running pods see it? The two ways in behave differently:
  - the mounted file: the kubelet rewrites it at its next synchronisation, about a minute later. The program must read the file again to notice.
  - the environment variable: set when the container started, never changes afterwards

```
$ kubectl exec deployment/api-staging -- cat /config/LOG_LEVEL        # 73 s later
debug
$ kubectl exec deployment/api-staging -- env | grep LOG_LEVEL
LOG_LEVEL=info
```

- So with `envFrom` a changed setting needs new pods. `rollout restart` replaces them one by one with the same template; the new ones read the new values.

```
$ kubectl rollout restart deployment/api-staging
deployment.apps/api-staging restarted
$ kubectl exec deployment/api-staging -- env | grep LOG_LEVEL
LOG_LEVEL=debug
```

A ConfigMap that is not there

- The same `k8s/api.yaml` deployed to dev, where no ConfigMap `api-dev` exists
- The ConfigMap is referenced by name. If it does not exist, the new pod waits in `ContainerCreating` until it does.
- The old pod keeps serving. The rollout is stuck, not broken.

```
$ kubectl get pods -l app=api-dev
NAME                       READY   STATUS              RESTARTS   AGE
api-dev-6c45974b5c-72ckv   1/1     Running             0          56s
api-dev-866c7b67c6-7pbnh   0/1     ContainerCreating   0          12s
$ kubectl describe pod api-dev-866c7b67c6-7pbnh | tail -1
  Warning  FailedMount  10s (x6 over 26s)  kubelet  MountVolume.SetUp failed for volume "config" : configmap "api-dev" not found
```

- Create the ConfigMap and the pod starts by itself:

```
$ kubectl create configmap api-dev --from-literal=ENVIRONMENT=dev
configmap/api-dev created
$ curl -s http://000007test-dev.k8s.digivader.com/info
{"environment":"dev"}
```

- With `envFrom` alone and no volume the status is `CreateContainerConfigError`; the same wait, the same fix

ConfigMap and Secret

- ConfigMap and Secret do the same job: hold values a pod reads at start. The difference is what goes in which.
- Both are read the same two ways: `envFrom` / `valueFrom`, or a volume
- ConfigMap: settings that anyone who can see the namespace may read; shown as plain text; goes into git as a manifest
- Secret: passwords, tokens, keys; shown base64-encoded; access kept narrow; never in git
- Both up to 1 MiB
- A ConfigMap per environment: `api-dev`, `api-staging`. The Deployment names it with the same placeholder as everything else.

Load

- The next thing that happens to a running service: the load grows. Does anything in the cluster react? Staging, idle, two pods:

```
$ kubectl top pods -l app=api-staging
NAME                           CPU(cores)   MEMORY(bytes)
api-staging-6b497cf865-c9cc6   5m           40Mi
api-staging-6b497cf865-kwlkb   6m           40Mi
```

- Load, made by hand: a pod inside the namespace that requests the service in four endless loops

```
$ kubectl run load --image=busybox:1.37 --restart=Never -- sh -c \
   'for i in 1 2 3 4; do (while true; do wget -q -O /dev/null http://api-staging/api/boardgames; done) & done; wait'
pod/load created
```

- `--image=busybox:1.37` : a 5 MB image with a shell and `wget` , enough to send requests and nothing more
- `--restart=Never` : a plain pod, not started again when its command ends
- `(while true; do wget …; done) &` : one endless loop of requests to the Service, in the background; four of them

```
$ kubectl top pods -l app=api-staging
NAME                           CPU(cores)   MEMORY(bytes)
api-staging-6b497cf865-c9cc6   200m         41Mi
api-staging-6b497cf865-kwlkb   235m         41Mi
$ kubectl get deployment api-staging
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
api-staging   2/2     2            2           19m
```

- The pods request 25m of CPU and use 200m and 235m, close to their limit of 250m
- `replicas: 2` is a number in a file. Nothing in the cluster reacts to the load.

Scaling

- Two things can change when the load grows:
  - **vertical scaling**: the same pods get more CPU and memory → higher requests and limits, the pods are restarted
  - **horizontal scaling**: more pods of the same size → a higher `replicas`
- Horizontal scaling needs a service that keeps nothing in its own memory between two requests: any pod can answer any request. Ours keeps everything in the database.
- Changing `replicas` by hand needs a person who watches the load
- An automatic change needs two things: a number that measures the load, and a rule

The number

- The **metrics server** is a component of the cluster that collects, from every node, how much CPU and memory every pod uses right now. `kubectl top` shows its numbers.
- `5m` is five thousandths of one CPU core
- **Utilisation** = use divided by request. A pod that requests 25m and uses 5m is at 20 %.
- A pod can use more than its request, up to its limit, so utilisation can be far above 100 %
- The request is the base of the percentage. A container without a CPU request has no utilisation.

The rule: HorizontalPodAutoscaler

- A **HorizontalPodAutoscaler** (HPA) is an object that names a Deployment, a metric, a target value and a range of replicas
- A controller reads the metric every 15 s and sets the Deployment's `replicas`:

```
wanted replicas = current replicas × current utilisation / target utilisation     (rounded up)
```

Current replicas	Utilisation	Target	Wanted
2	20 %	60 %	2 × 20 / 60 = 0.67 → 1, raised to the minimum
2	60 %	60 %	2
2	150 %	60 %	2 × 150 / 60 = 5
4	30 %	60 %	4 × 30 / 60 = 2

- The utilisation is the average over the Deployment's pods
- The result is kept between the HPA's minimum and maximum

An HPA manifest

- The rule for staging as an object, `hpa.yaml` : between two and five pods, at 60 % of their CPU request on average

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: api-staging
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: api-staging
  minReplicas: 2
  maxReplicas: 5
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 60
```

- `scaleTargetRef` : the object whose `replicas` this HPA sets
- `minReplicas` , `maxReplicas` : never fewer, never more
- `type: Resource` , `name: cpu` : the metric, CPU use from the metrics server
- `averageUtilization: 60` : the target, the pods' average CPU use at 60 % of their CPU request

An HPA for staging

- Applied while staging is idle:

```
$ kubectl apply -f hpa.yaml
horizontalpodautoscaler.autoscaling/api-staging created
$ kubectl get hpa api-staging
NAME          REFERENCE                TARGETS        MINPODS   MAXPODS   REPLICAS   AGE
api-staging   Deployment/api-staging   cpu: 22%/60%   2         5         2          31s
```

- `TARGETS` : current utilisation / target; `REPLICAS` : what the HPA last set
- `cpu: <unknown>/60%` in the first seconds: the metrics server has no number for the pods yet
- 22 % with two pods and a target of 60 %: the formula asks for one, the minimum keeps two

The load returns

- The load pod started again. `kubectl get hpa api-staging` every 15 s:

After the load began	`TARGETS`	`REPLICAS`
15 s	`cpu: 22%/60%`	2
30 s	`cpu: 228%/60%`	2
45 s	`cpu: 580%/60%`	4
60 s	`cpu: 840%/60%`	5
75 s	`cpu: 479%/60%`	5
90 s to 150 s	`cpu: 406%/60%` to `434%/60%`	5

```
$ kubectl describe hpa api-staging | tail -6
  ScalingLimited  True    TooManyReplicas      the desired replica count is more than the maximum replica count
Events:
  Type    Reason             Age   From                       Message
  Normal  SuccessfulRescale  2m4s  horizontal-pod-autoscaler  New size: 4; reason: cpu resource utilization (percentage of request) above target
  Normal  SuccessfulRescale  109s  horizontal-pod-autoscaler  New size: 5; reason: cpu resource utilization (percentage of request) above target
```

What happened, and why

- **Nothing for 15 s.** The metrics server collects at intervals; the first reading that contained the load arrived after that.
- **From 2 to 4, not to 5.** The first computation that saw the load saw a window that was only partly under load, and asked for 4.
- **Then 5.** The next reading was 580 %: 2 × 580 / 60 asks for 20. The answer is the maximum: `TooManyReplicas`.
- **Then about 410 %, and it stays.** Five pods share the same four loops; each uses about 100m, four times its request. The HPA would add more; it may not.
- Every step is an event on the HPA, with its reason. `kubectl describe hpa` is where to look when the number is not what you expect.

The pods

- The five pods the HPA asked for: where they run and what they use

```
$ kubectl get pods -l app=api-staging -o wide
NAME                           READY   STATUS    RESTARTS   AGE     IP             NODE
api-staging-6b497cf865-2rxmk   1/1     Running   0          109s    10.42.7.213    k3s-worker-1
api-staging-6b497cf865-kfvv5   1/1     Running   0          5m51s   10.42.9.250    k3s-worker-5
api-staging-6b497cf865-klk84   1/1     Running   0          2m4s    10.42.3.185    k3s-worker-4
api-staging-6b497cf865-kwlkb   1/1     Running   0          6m3s    10.42.5.199    k3s-worker-6
api-staging-6b497cf865-vns7d   1/1     Running   0          2m4s    10.42.10.186   k3s-worker-3
$ kubectl top pods -l app=api-staging
NAME                           CPU(cores)   MEMORY(bytes)
api-staging-6b497cf865-2rxmk   111m         41Mi
api-staging-6b497cf865-kfvv5   93m          41Mi
api-staging-6b497cf865-klk84   100m         41Mi
api-staging-6b497cf865-kwlkb   103m         41Mi
api-staging-6b497cf865-vns7d   106m         41Mi
```

- Five pods on five workers: the scheduler spreads the copies of one Deployment
- The same four loops, now shared by five pods instead of two: about 100m each instead of 200m, still four times the request
- The HPA only sets a number. What caps the pods that actually run:
  - its own `maxReplicas`
  - the namespace's quota: a pod over it is not created
  - room on the nodes: a pod that fits nowhere stays `Pending`
- The HPA sees none of this; it keeps the number the formula gave
- The namespace's CPU requests are used up at exactly five: a sixth pod would not have been created

```
$ kubectl get resourcequota student-quota
NAME            REQUEST                                                                                     LIMIT
student-quota   … pods: 12/20, requests.cpu: 500m/500m, requests.memory: 1216Mi/2Gi …   limits.cpu: 4/6, limits.memory: 3840Mi/6Gi
```

How the Service got the new pods

- A Service has a selector. For every Service the cluster keeps a list of the pods that match it right now, with each pod's address and whether it is ready: an **EndpointSlice**

```
$ kubectl get endpointslices -l kubernetes.io/service-name=api-staging
NAME                ADDRESSTYPE   PORTS   ENDPOINTS                                 AGE
api-staging-4h8fk   IPv4          8000    10.42.5.199,10.42.9.250,10.42.3.185 + 2 more...   10m
$ kubectl get endpointslices -l kubernetes.io/service-name=api-staging \
    -o jsonpath='{range .items[*].endpoints[*]}{.addresses[0]} ready={.conditions.ready} node={.nodeName}{"\n"}{end}'
10.42.5.199 ready=true node=k3s-worker-6
10.42.9.250 ready=true node=k3s-worker-5
10.42.3.185 ready=true node=k3s-worker-4
10.42.10.186 ready=true node=k3s-worker-3
10.42.7.213 ready=true node=k3s-worker-1
```

- `-l kubernetes.io/service-name=api-staging` : the slices that belong to this Service
- A pod the HPA adds is in the list as soon as its readiness probe passes. A pod whose probe fails stays in the list with `ready=false` and gets no requests.
- A Service that answers nothing has an empty list: its selector matches no pod

Service types

- A Service's `type` says from where it can be reached. Three types, each the one before plus one thing.
- **`ClusterIP`**: a virtual address from the cluster's own range and a name, `api-staging`. Reachable inside the cluster only. Ours; the load pod and the Ingress controller reach it this way.
- **`NodePort`**: a `ClusterIP`, plus one port between 30000 and 32767 opened on every node and forwarded to it. Reachable from outside at any node's address and that port, whether or not that node runs a pod of the Service.
  - nothing to install: the simplest way out, used on bare machines and in test clusters
  - the port is not 80 or 443; node addresses change; every node's firewall opens for it

```
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 8000
    nodePort: 31234      # optional; the cluster picks one otherwise
```

- **`LoadBalancer`**: a `NodePort`, plus a load balancer the cluster orders from the cloud it runs in, placed in front of the nodes. Reachable at one address of its own, on the port you choose.
  - one cloud balancer per Service, with its own public address and its own price; on fixed machines there is nobody to order it from

Why ours stay ClusterIP

- **Traefik** is the Ingress controller of the course cluster: a web proxy installed with the cluster by the administrator, in `kube-system`. It reads every Ingress in every namespace and routes by host name.
- It is the one `LoadBalancer` Service on the cluster, on ports 80 and 443 of every node. The DNS name `k8s.digivader.com` points at the nodes; Traefik answers.
- It balances already: a request for your host name goes to your Service's `ClusterIP`, which spreads it over your pods. One door for everyone, nothing to expose per Service.
- The student quota allows no `NodePort` and no `LoadBalancer`: the administrator's door is the only one
- When you still meet them:
  - a manifest written for a cloud cluster, with `type: LoadBalancer`
  - a cluster with no Ingress controller
  - a protocol that is not HTTP, which an Ingress cannot route: a database, a game server

The load stops

- The load pod deleted. Does the HPA come down as fast as it went up?

```
$ kubectl delete pod load --now
```

After the load stopped	`TARGETS`	`REPLICAS`
30 s	`cpu: 350%/60%`	5
60 s	`cpu: 20%/60%`	5
90 s to 300 s	`cpu: 20%/60%` to `22%/60%`	5
330 s	`cpu: 22%/60%`	4
360 s	`cpu: 22%/60%`	2

```
$ kubectl describe hpa api-staging | grep -A6 '^Events' | tail -2
  Normal  SuccessfulRescale  107s   horizontal-pod-autoscaler  New size: 4; reason: All metrics below target
  Normal  SuccessfulRescale  92s    horizontal-pod-autoscaler  New size: 2; reason: All metrics below target
```

- A removed pod cannot be put back in an instant: schedule, pull, start, pass the readiness probe
- So the HPA scales down slowly: it uses the highest number it computed in the last 5 minutes
- The 5 minutes are the default of `behavior.scaleDown.stabilizationWindowSeconds` in the HPA's `spec`

`replicas:` and the HPA

- Two writers now set the same field, the Deployment's `spec.replicas`:
  - the HPA writes the number from its formula, every 15 s when it differs
  - `kubectl apply` of the Deployment manifest writes the number in the file, at every deploy
- A deploy during load puts the Deployment back to the file's number; the HPA raises it again seconds later; in between the service has too few pods
- THE RULE: a Deployment that an HPA manages has no `replicas:` line in its manifest. `kubectl apply` then leaves the field as it is.
- What an HPA does not do:
  - add nodes: when the nodes are full, new pods stay `Pending`. A **cluster autoscaler** is a separate component that orders nodes from the cloud; a cluster on fixed machines has none.
  - scale the database: more service pods send more queries to the same database
  - see load that is not CPU: a service waiting for a slow database has low CPU use under high load. Other metrics, such as requests per second, need a component that provides them.

A node is emptied

- Before an upgrade or a repair an administrator empties a node: `kubectl drain <node>`
- Drain does not delete pods. For each pod it asks the API server for an **eviction**: end this pod. The API server may refuse, if a rule says too few pods would stay.
- The Deployment starts a replacement elsewhere. Between the eviction and the replacement's readiness probe there is one pod fewer.
- Staging has two pods on two nodes. Both nodes drained shortly after each other: the second pod is evicted while the replacement of the first is not ready yet. For some seconds no pod answers.
- The same for an autoscaler that removes a node, and for the cloud's own maintenance of a virtual machine. All of these go through the API; a node that loses power does not.

PodDisruptionBudget

- A **PodDisruptionBudget** (PDB) says, for the pods matching a selector, how many must stay available while pods are evicted. The API server refuses an eviction that would go below it.

```yaml
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: api-staging
spec:
  minAvailable: 1
  selector:
    matchLabels:
      app: api-staging
```

```
$ kubectl apply -f pdb.yaml
poddisruptionbudget.policy/api-staging created
$ kubectl get pdb api-staging
NAME          MIN AVAILABLE   MAX UNAVAILABLE   ALLOWED DISRUPTIONS   AGE
api-staging   1               N/A               1                     5s
```

- `MIN AVAILABLE 1` : what we wrote, `minAvailable: 1`
- `MAX UNAVAILABLE N/A` : the other way to write the same rule, `maxUnavailable: 1` instead of `minAvailable`. A PDB has one or the other, never both. We did not use it.
- `ALLOWED DISRUPTIONS 1` : not set by anyone. The API server computes it: 2 pods ready minus 1 that must stay = 1 eviction it would allow right now. It changes as pods come and go.

Two evictions in a row

- An eviction is a small object sent to the pod's `eviction` address in the API; drain sends one per pod. By hand, for both pods:

```
$ cat evict-api-staging-6b497cf865-kfvv5.json
{"apiVersion":"policy/v1","kind":"Eviction","metadata":{"name":"api-staging-6b497cf865-kfvv5","namespace":"000007test"}}
$ kubectl create --raw /api/v1/namespaces/000007test/pods/api-staging-6b497cf865-kfvv5/eviction -f evict-api-staging-6b497cf865-kfvv5.json
{"kind":"Status","apiVersion":"v1","metadata":{},"status":"Success","code":201}
$ kubectl create --raw /api/v1/namespaces/000007test/pods/api-staging-6b497cf865-kwlkb/eviction -f evict-api-staging-6b497cf865-kwlkb.json
Error from server (TooManyRequests): Cannot evict pod as it would violate the pod's disruption budget.
$ kubectl get pods -l app=api-staging
NAME                           READY   STATUS              RESTARTS   AGE
api-staging-6b497cf865-c9cc6   0/1     ContainerCreating   0          0s
api-staging-6b497cf865-kfvv5   1/1     Terminating         0          13m
api-staging-6b497cf865-kwlkb   1/1     Running             0          13m
$ kubectl get pdb api-staging
NAME          MIN AVAILABLE   MAX UNAVAILABLE   ALLOWED DISRUPTIONS   AGE
api-staging   1               N/A               0                     6s
```

- `kubectl create --raw <path> -f <file>` : send the file to that path of the API as it is
- The first eviction is accepted. The second is refused: the only ready pod left is the one it names.
- 25 s later the replacement was ready and `ALLOWED DISRUPTIONS` was 1 again. A drain retries until then.

What a PDB covers

- Yes: `kubectl drain`, an autoscaler removing a node, the cloud's maintenance. They evict.
- No: `kubectl delete pod`, a deletion is not an eviction
- No: a rolling update; the Deployment's own `maxUnavailable` and `maxSurge` rule it
- No: a node that fails; nobody asked
- `minAvailable: 1` with one replica, or `minAvailable: 2` with two → the pod can never be evicted, a drain waits forever, the administrator deletes the pod by hand
- A selector that matches no pod → nothing

The files have drifted

- Everything since the ConfigMap was made by hand, in the cluster, in staging. The repository and the pipeline know nothing of it:

```
$ kubectl get configmaps,hpa,pdb
NAME                          DATA   AGE
configmap/api-dev             1      19m
configmap/api-staging         2      23m

NAME                                                 REFERENCE                 TARGETS        MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/api-staging   Deployment/api-staging   cpu: 22%/60%   2         5         2          6m8s

NAME                                       MIN AVAILABLE   MAX UNAVAILABLE   ALLOWED DISRUPTIONS   AGE
poddisruptionbudget.policy/api-staging   1               N/A               1                     8m28s
```

- The next deploy of staging puts `replicas: 2` back into the Deployment. The HPA fights it.
- They have to go into the repository. The repository has two files and four placeholders.

What the placeholders cannot do

- `envsubst` swaps `$ENV` for text and nothing else. Every file in `k8s/` goes to both environments as it is, with the words changed.
- What dev and staging need to differ in, and a placeholder cannot do:
  - `hpa.yaml` for staging, not for dev: there is no "skip this file in dev"
  - `replicas: 2` for dev, no such line for staging: there is no "skip this line in staging"
  - two keys in staging's ConfigMap, one in dev's: same lines, same number of keys
- An object removed from the files stays in the cluster: `kubectl apply` deletes nothing
- What is installed, since when, from which commit: recorded nowhere. Going back = find the old tag, run the pipeline again.
- The manifests need to be a package: one set of templates, one set of values per environment, installed as one thing with a history.

Helm

- **Helm** is a program that installs applications into a Kubernetes cluster from packages
- **chart**: the package, a directory with manifest **templates** and a file of default values
- **values**: the settings that fill the templates for one installation
- **release**: one installation of a chart in a namespace, with a name and a numbered history
- One chart installed twice with two sets of values = two releases: `dev` and `staging`

```
chart + values  ⟶  manifests  ⟶  cluster
                   recorded as revision N of the release
```

- Helm runs where `kubectl` runs and uses the same kubeconfig. Nothing is installed in the cluster for it.

A chart

- The manifests of the repository, moved into a chart, plus one values file per environment:

```
chart/
├── Chart.yaml              name and version of the chart
├── values.yaml             the default values
└── templates/
    ├── configmap.yaml
    ├── deployment.yaml
    ├── service.yaml
    ├── ingress.yaml
    ├── db.yaml             the database: Service and StatefulSet
    └── hpa.yaml            HPA and PDB
values-dev.yaml             what differs in dev
values-staging.yaml         what differs in staging
```

```yaml
# chart/Chart.yaml
apiVersion: v2
name: boardgames
description: The boardgames service with its database
version: 0.1.0
```

- `version` is the version of the chart itself; it changes when the templates change

A template

- A template is a manifest with expressions in `{{ }}`. Helm replaces every expression with its value and sends the result to the cluster. `chart/templates/service.yaml`:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: api-{{ .Release.Name }}
spec:
  selector:
    app: api-{{ .Release.Name }}
  ports:
  - port: 80
    targetPort: 8000
```

- The first word of an expression says where the value comes from:
  - `.Release`: the install command. `{{ .Release.Name }}` is the release's name, `dev` or `staging`. It replaces `$ENV`.
  - `.Values`: the values files. `{{ .Values.host }}` is the key `host`; `{{ .Values.image.repository }}` is `repository` under `image`. They replace the other placeholders.
  - `.Chart`: `Chart.yaml`. `{{ .Chart.Name }}` is its `name`.

Values

- The values fill the templates. Helm looks for each value in this order and takes the first it finds:
  1. `--set key=value` on the command line
  2. a file given with `-f`, `values-dev.yaml`
  3. `values.yaml` in the chart, the defaults
- `chart/values.yaml`, the defaults:

```yaml
image:
  repository: supax/iti8801-a5-reference
  tag: ""                  # the pipeline sets it: --set image.tag=<commit>
host: ""                   # the Ingress host
replicas: 1                # used when autoscaling is off
config:                    # every key becomes an environment variable of the service
  ENVIRONMENT: ""
autoscaling:
  enabled: false
  minReplicas: 2
  maxReplicas: 5
  cpuUtilization: 60
```

- The two environment files, given with `-f`, hold only what differs from the defaults:

```yaml
# values-dev.yaml
host: 000007test-dev.k8s.digivader.com
config:
  ENVIRONMENT: dev
```

```yaml
# values-staging.yaml
host: 000007test-staging.k8s.digivader.com
config:
  ENVIRONMENT: staging
autoscaling:
  enabled: true
```

Only in staging

- The HPA and the PDB exist in staging and not in dev; `replicas:` the other way round. A condition in the template:

```
# deployment.yaml
spec:
  {{- if not .Values.autoscaling.enabled }}
  replicas: {{ .Values.replicas }}
  {{- end }}
```

```
# hpa.yaml
{{- if .Values.autoscaling.enabled }}
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
…
{{- end }}
```

- `{{- if … }} … {{- end }}`: the lines between are part of the result only when the condition holds; the `-` removes the empty line the expression would leave
- With `autoscaling.enabled: true` the Deployment has no `replicas:` line and the HPA and the PDB exist; with `false` it is the other way round. The two things `envsubst` could not do.
- A second check, deeper in the same `deployment.yaml`: a value that must be given

```
    containers:
    - name: api
      image: {{ .Values.image.repository }}:{{ required "image.tag is not set" .Values.image.tag }}
```

- `{{ required "message" <value> }}`: stops with the message when the value is empty. The pipeline sets `image.tag`; a run without it stops here instead of deploying `:latest` or nothing.

The ConfigMap as a loop

- Two keys in one environment, one in the other, more later. A loop over whatever the values hold:

```
# configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: api-{{ .Release.Name }}
data:
  {{- range $key, $value := .Values.config }}
  {{ $key }}: {{ $value | quote }}
  {{- end }}
```

- `{{ range $key, $value := .Values.config }} … {{ end }}` : repeats the lines between once for every key under `config`
- `{{ $value | quote }}` : the value in quotes, a ConfigMap value must be a string
- A setting added under `config` in a values file becomes a key of the ConfigMap. The template does not change. The third thing `envsubst` could not do.

Rendering without a cluster

- `helm template` prints the manifests a release would get and contacts no cluster:

```
$ helm template dev chart -f values-dev.yaml --set image.tag=0fa0b979
---
# Source: boardgames/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: api-dev
data:
  ENVIRONMENT: "dev"
---
# Source: boardgames/templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-dev
spec:
  replicas: 1
…
```

- `helm template dev chart` : fill the templates in the folder `chart` with values and print the manifests, as for a release named `dev`
- `-f values-dev.yaml` : take values from this file; `--set image.tag=0fa0b979` : and this one value from the command line

```
$ helm template dev chart -f values-dev.yaml
Error: execution error at (boardgames/templates/deployment.yaml:21:49): image.tag is not set
$ helm lint chart -f values-dev.yaml --set image.tag=0fa0b979
1 chart(s) linted, 0 chart(s) failed
```

Installing a release

- `helm upgrade --install` creates the release when it does not exist and updates it when it does: one command for the first deploy and every later one

```
$ helm upgrade --install dev chart -f values-dev.yaml --set image.tag=0fa0b979
Release "dev" does not exist. Installing it now.
Error: unable to continue with install: Service "db-dev" in namespace "000007test" exists and cannot be imported
  into the current release: invalid ownership metadata; label validation error: missing key
  "app.kubernetes.io/managed-by": must be set to "Helm"; …
```

- Helm refuses to take over objects it did not create. The week 7 objects of dev are deleted by hand first, once.
- The data is in the volume claim, `data-db-dev-0`, not in the StatefulSet. The StatefulSet only describes the pod, and deleting it leaves the claim. Helm's StatefulSet, same name, same template, mounts the same claim. The data stays.

```
$ kubectl delete deployment,service,ingress api-dev
$ kubectl delete statefulset,service db-dev
$ kubectl get pvc data-db-dev-0
NAME            STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
data-db-dev-0   Bound    pvc-2ba786ef-88a7-4310-8c7e-8915fcb390cf   1Gi        RWO            local-path     35m
$ helm upgrade --install dev chart -f values-dev.yaml --set image.tag=0fa0b979 --wait --timeout 2m
Release "dev" does not exist. Installing it now.
NAME: dev
NAMESPACE: 000007test
STATUS: deployed
REVISION: 1
```

- `--wait`: Helm waits until the new pods are ready; `--timeout 2m`: the wait ends as a failure after two minutes

What Helm keeps

- Helm records every release it made. The record is in the cluster, in two places.
- On every object it made, a label and two annotations: made by Helm, for release `dev`, in this namespace. This is what the refusal on the previous page checked.

```
$ kubectl get deployment api-dev -o jsonpath='{.metadata.labels}{"\n"}{.metadata.annotations}{"\n"}'
{"app.kubernetes.io/managed-by":"Helm"}
{"deployment.kubernetes.io/revision":"1","meta.helm.sh/release-name":"dev","meta.helm.sh/release-namespace":"000007test"}
```

- `-o jsonpath=…`: print only these two fields of the object
- One Secret per revision, `sh.helm.release.v1.dev.v1`: the rendered manifests and the values of that revision, in the namespace

```
$ kubectl get secrets
NAME                         TYPE                 DATA   AGE
db-dev                       Opaque               1      48m
db-staging                   Opaque               1      48m
sh.helm.release.v1.dev.v1    helm.sh/release.v1   1      12s
```

- `helm list`, `helm history`, `helm get values` read these Secrets and nothing else. Any machine with the kubeconfig sees the same.

```
$ helm list
NAME   NAMESPACE    REVISION   UPDATED                STATUS     CHART
dev    000007test   1          2026-10-05 23:02:26    deployed   boardgames-0.1.0
$ helm get values dev
USER-SUPPLIED VALUES:
config:
  ENVIRONMENT: dev
host: 000007test-dev.k8s.digivader.com
image:
  tag: 0fa0b979
```

A change, a history, a rollback

- The same change as before, `LOG_LEVEL=debug`, now as a value of the release instead of a patch on the ConfigMap:

```
$ helm upgrade dev chart --reuse-values --set config.LOG_LEVEL=debug --wait
Release "dev" has been upgraded. Happy Helming!
REVISION: 3
$ kubectl get pods -l app=api-dev
NAME                       READY   STATUS        RESTARTS   AGE
api-dev-67f454868b-s5wrw   1/1     Running       0          13s
api-dev-6b5d7fc596-mb7br   1/1     Terminating   0          114s
$ kubectl exec deployment/api-dev -- env | grep -E 'LOG_LEVEL|ENVIRONMENT'
ENVIRONMENT=dev
LOG_LEVEL=debug
```

- `--reuse-values`: start from the values of the release as it is now. The new key under `config` became a ConfigMap key and an environment variable.
- The pod was replaced with no `rollout restart`: the pod template carries a checksum of the rendered ConfigMap, `checksum/config: {{ include (print $.Template.BasePath "/configmap.yaml") . | sha256sum }}`. A changed value changes the checksum, the checksum changes the template, the Deployment replaces its pods.

```
$ helm rollback dev 2 --wait
Rollback was a success! Happy Helming!
$ helm history dev
REVISION   UPDATED                    STATUS       CHART              DESCRIPTION
1          Mon Oct  5 23:02:26 2026   superseded   boardgames-0.1.0   Install complete
2          Mon Oct  5 20:04:20 2026   superseded   boardgames-0.1.0   Upgrade complete
3          Mon Oct  5 23:06:02 2026   superseded   boardgames-0.1.0   Upgrade complete
4          Mon Oct  5 23:06:16 2026   deployed     boardgames-0.1.0   Rollback to 2
$ kubectl exec deployment/api-dev -- env | grep -E 'LOG_LEVEL|ENVIRONMENT'
ENVIRONMENT=dev
```

- `helm rollback dev 2` applies the manifests of revision 2 again, all objects at once, as a new revision. Revision 2 was made by the pipeline, on a machine whose clock is set to UTC.

An upgrade that cannot finish

- `--rollback-on-failure` makes Helm go back to the last working revision by itself when the wait ends in a failure. An image tag that does not exist:

```
$ helm upgrade dev chart --reuse-values --set image.tag=nosuchtag --wait --timeout 40s --rollback-on-failure
Error: UPGRADE FAILED: release dev failed, and has been rolled back due to rollback-on-failure being set:
  resource Deployment/000007test/api-dev not ready. status: InProgress, message: Pending termination: 1
  context deadline exceeded
$ helm history dev | tail -2
5          Mon Oct  5 23:06:29 2026   failed       boardgames-0.1.0   Upgrade "dev" failed: resource Deployment/000007test/api-dev not ready. …
6          Mon Oct  5 23:07:10 2026   deployed     boardgames-0.1.0   Rollback to 4
$ kubectl get pods -l app=api-dev
NAME                       READY   STATUS        RESTARTS   AGE
api-dev-6694d7cd6b-q54f2   0/1     Terminating   0          41s
api-dev-6b5d7fc596-b9qhs   1/1     Running       0          55s
```

- The command ended with a code other than 0, the failed revision is in the history, and the release is back at a state that works
- Helm 3, the previous version and the one most tutorials still show, calls this flag `--atomic`; it does the same thing

The deploy job with Helm

- The deploy job of the pipeline: the loop with `envsubst`, the `kubectl apply` and the `kubectl rollout status` become one line

```
.deploy:
  stage: deploy
  image: alpine/k8s:1.36.5
  script:
    - helm upgrade --install $ENV chart -f values-$ENV.yaml --set image.tag=$CI_COMMIT_SHORT_SHA --wait --timeout 2m

deploy-dev:
  extends: .deploy
  variables: { ENV: dev }
  rules:
    - if: $CI_COMMIT_BRANCH == "main"

deploy-staging:
  extends: .deploy
  variables: { ENV: staging }
  rules:
    - if: $CI_COMMIT_TAG
```

```
$ helm upgrade --install $ENV chart -f values-$ENV.yaml --set image.tag=$CI_COMMIT_SHORT_SHA --wait --timeout 2m
Release "dev" has been upgraded. Happy Helming!
NAME: dev
STATUS: deployed
REVISION: 2
Job succeeded
```

- The image `alpine/k8s:1.36.5` contains Helm 4.3. The job took 22 s.

Both releases

- The same `kubectl get` as before the chart, now after the tag pipeline has installed staging:

```
$ helm list
NAME      NAMESPACE    REVISION   UPDATED                    STATUS     CHART
dev       000007test   2          2026-10-05 20:04:20 UTC   deployed   boardgames-0.1.0
staging   000007test   1          2026-10-05 20:05:39 UTC   deployed   boardgames-0.1.0
$ kubectl get deployments,hpa,pdb,configmaps
NAME                          READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/api-dev       1/1     1            1           3m35s
deployment.apps/api-staging   1/2     2            1           21s

NAME                                          REFERENCE                TARGETS              MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/api-staging   Deployment/api-staging   cpu: <unknown>/60%   2         5         1          21s

NAME                                    MIN AVAILABLE   MAX UNAVAILABLE   ALLOWED DISRUPTIONS   AGE
poddisruptionbudget.policy/api-staging   1               N/A               0                     22s

NAME                          DATA   AGE
configmap/api-dev             1      3m36s
configmap/api-staging         1      23s
configmap/kube-root-ca.crt    1      26h
```

- The same chart, two values files: dev has `replicas: 1` and no HPA; staging has no `replicas:` line, an HPA and a PDB
- Everything the hand-made staging had is now in the repository, per environment, with a history
- Staging shows `1/2` and the HPA `REPLICAS 1`: a Deployment with no `replicas:` line starts with one pod. 21 s after the install the HPA is raising it to its minimum of two; until then the PDB allows no disruption, one pod healthy and one must stay.

What Helm removes, what it leaves

- `kubectl apply` deletes nothing. Helm knows what a release created, so it can:
  - a template deleted from the chart, or its condition no longer true → the object is deleted at the next upgrade
  - `helm uninstall <release>` → every object of the release, and its history
- What stays:
  - a volume claim: Helm made the StatefulSet, the StatefulSet made the claim, and a deleted StatefulSet leaves its claims behind
  - a Secret made by hand: Helm never saw it
  - an object changed with `kubectl edit`: the change stays until the next upgrade touches the same field
- A release and its files in git describe the same thing only when every change goes through the chart

Charts by others

- Charts are also how software from others is installed: ingress controllers, databases, monitoring. They live in the same registries as images.

```
$ helm show values oci://registry-1.docker.io/bitnamicharts/postgresql | less
$ helm install db oci://registry-1.docker.io/bitnamicharts/postgresql -f my-values.yaml
```

- `helm show values <chart>`: what can be set, the chart's `values.yaml` with its comments
- `helm install <release> <chart> -f my-values.yaml`: installed with your values over its defaults, like our own chart
- Our database is a StatefulSet we wrote, to see what is in one. In a project you would install it this way, or use a managed database.
- A chart from somebody else creates whatever its templates say, with your credential: `helm template` shows all of it before anything is installed
- Bitnami, the catalogue every tutorial names, moved its versioned images behind a paid product in September 2025: the chart installs, the image does not pull. `helm template` also shows which images a chart pulls; read that before you trust a tutorial.
- **Kustomize** is the other common tool for the same problem: no templates, a base set of plain manifests and per environment an **overlay** that lists changes to it. Built into `kubectl`: `kubectl apply -k <directory>`.

Push

- Everything in this lecture, and in the pipeline of the last one, is **push**: a pipeline outside the cluster holds a cluster credential and applies
- Git says what should run. The pipeline makes it so, and only when it runs: at a push to `main`, at a tag.
- A change made by hand in the cluster stays until the next deploy. Nobody notices until then.
- The credential lives outside the cluster, as a CI/CD variable. Whoever reads the variable has the cluster.
- Which commit runs where is recorded in the pipeline's history and in Helm's, not in git

Pull: GitOps

- The reverse is **pull**: a program inside the cluster watches the repository and applies what it finds. The practice is called **GitOps**.
- Git is the single source of truth: what should run is the chart and the values files at the head of a branch, nothing else
- The program checks continuously, not only at a push: every few minutes it renders the chart and compares the result with the cluster
- A difference between git and the cluster is corrected in git's favour: an object changed by hand is put back, an object missing is created
- No cluster credential leaves the cluster. The pipeline builds, tests and scans the image, then commits the new tag into the values file; the deploy is the program noticing the commit.
- The history is the git history. Going back is a revert; the program applies it like any other commit.

Argo CD

- **Argo CD** is a program that runs in the cluster and does the pull. **Flux** is the same without a web page.
- One object per environment, **Application**: which repository, which path, which values file, which namespace
- It renders the chart, compares the result with the cluster, and shows on a web page whether the cluster matches git and whether the pods are up
- Per Application you choose: apply a change in git by itself or wait for a person to click; undo changes made by hand or leave them; delete objects removed from git or keep them
- Nothing else changes: the chart and the values files are the same, the pipeline builds and tests the image as before

The commands of this lecture

Command	Does
`kubectl create configmap <name> --from-literal=K=v`	a ConfigMap from values; `--from-file=<file>` for a whole file
`kubectl rollout restart deployment/<name>`	replace the pods one by one, same template
`kubectl top pods`, `kubectl get hpa`, `kubectl describe hpa <name>`	current use; the autoscalers; one with its conditions and events
`kubectl get endpointslices -l kubernetes.io/service-name=<service>`	the pods behind a Service
`kubectl get pdb`	the disruption budgets with the evictions allowed now
`helm template <release> <chart> -f <file> --set k=v`	render, no cluster
`helm lint <chart> -f <file>`	check a chart
`helm upgrade --install <release> <chart> -f <file> --wait`	install or update
`helm list`, `helm history <release>`, `helm get values <release>`	the releases; one's revisions; its values
`helm rollback <release> <revision>`, `helm uninstall <release>`	back to a revision; remove the release



Questions