AIS · KUBERNETES HANDS-ON

# Kubernetes Hands-On Field Guide

Every command from Part 1, with what it does.

Episodes 1-15 · updated 2026-11-05 · the latest version is always at aislabs.ai/kubernetes

## EPISODE 1 Your first cluster heals itself

| | |
|---|---|
| `kind create cluster --name demo --wait 60s` | Create a local Kubernetes cluster inside Docker, and wait until it is ready. |
| `kubectl get nodes` | List the machines in the cluster and whether each one is Ready. |
| `kubectl get pods -n kube-system` | See the cluster's own parts (API server, etcd, scheduler, DNS) running as pods. |
| `kubectl create deployment web --image=nginx:alpine --replicas=3` | Ask for three copies of a web server. |
| `kubectl get pods -w` | Watch pods change state live. Ctrl+C stops watching. |
| `kubectl get deployment,replicaset,pods` | See the chain: a deployment owns a replica set, which owns the pods. |
| `kubectl delete pod POD_NAME` | Delete one pod. The replica set starts a replacement within seconds. |
| `kubectl delete pods -l app=web` | Delete every pod with that label. All of them are replaced. |
| `kubectl scale deployment web --replicas=5` | Change how many copies you want. |
| `kubectl delete deployment web --cascade=foreground` | Remove the app for real: nothing asks for the pods any more, so none come back. |
| `kind delete cluster --name demo` | Delete the whole cluster. |

## EPISODE 2 Docker containers explained

| | |
|---|---|
| `docker pull nginx:alpine` | Download an image: a packaged filesystem plus a default command. |
| `docker images` | List the images stored on this machine. |
| `docker run -d --name web -p 127.0.0.1:8080:80 nginx:alpine` | Start a container in the background, with port 8080 on the loopback address mapped to port 80 inside. |
| `docker ps` | List running containers. |
| `curl -s localhost:8080 \| head -n 4` | Fetch the page the container serves. |
| `docker logs --tail 2 web` | Show the last lines the container printed. |
| `docker exec web nginx -v` | Run a command inside the running container. |
| `docker inspect -f '{{.State.Status}}' web` | Read one field of everything Docker knows about the container. |
| `docker stop web` | Stop the container. It stays on disk, exited. |
| `docker ps -a` | List every container, stopped ones included. |
| `docker rm web` | Remove a stopped container. The image stays. |
| `docker rm -f blue green` | Stop and remove running containers in one step. |

## EPISODE 3 Dockerfile tutorial: build your own image

| | |
|---|---|
| `mkdir hello && cd hello` | Make a folder for the app and step into it. |
| `echo '<h1>Hello from my own image</h1>' > index.html` | Write the one-line web page the image will serve. |
| `cat > Dockerfile <<EOF (the 7 lines, then EOF)` | Write the Dockerfile: the recipe for the image, one instruction per line. |
| `docker build -t hello:1.0 .` | Build an image from the Dockerfile in this folder, tagged hello:1.0. The first build downloads the base image. |
| `docker images` | List the images on this machine: the new one sits under the name and tag you gave it. |
| `docker run -d --name hello -p 127.0.0.1:8000:8000 hello:1.0` | Start a container from the image in the background, with its port published on this machine only. |
| `curl -s localhost:8000` | Fetch the page the container serves. |
| `docker build -t hello:2.0 .` | After changing the page, build version 2: the steps that didn't change say CACHED. |
| `docker rm -f hello` | Stop and remove the container. The images stay. |
| `docker images hello` | List only the hello images: both versions are kept, each under its own tag. |

## EPISODE 4 Run your own image in Kubernetes

| | |
|---|---|
| `docker ps` | List running containers: the kind cluster's node and the local registry. |
| `kubectl get nodes` | Check that the cluster's node is Ready. |
| `docker build -t localhost:5001/hello:1.0 .` | Build the image with a tag that names the local registry. |
| `docker push localhost:5001/hello:1.0` | Push it to the local registry, where the cluster can pull it. |
| `kubectl create deployment hello --image=localhost:5001/hello:1.0` | Run your image in the cluster as a deployment. |
| `kubectl rollout status deployment/hello` | Wait until the deployment's pod is up. |
| `kubectl port-forward deployment/hello 8000:8000 > /dev/null 2>&1 &` | Forward local port 8000 to the pod, in the background. |
| `curl -s localhost:8000` | Fetch the page from the pod, through the forward. |
| `kubectl set image deployment/hello hello=localhost:5001/hello:9.9` | The mistake: point the deployment at a tag that was never pushed. |
| `kubectl events --types=Warning` | Show only warning events: the failed pull and the back-off. |
| `kubectl rollout undo deployment/hello` | The fix: roll back to the last version that worked. |
| `kubectl delete deployment hello` | Remove the deployment and its pods. |

## EPISODE 5 Kubernetes pods explained

| | |
|---|---|
| `kubectl run web --image=nginx:alpine` | Start a single pod from the nginx image. |
| `kubectl wait --for=condition=Ready pod/web --timeout=120s` | Wait until the pod is ready, for up to two minutes. |
| `kubectl get pods -o wide` | List pods with the node each one runs on and its own address in the cluster. |
| `kubectl describe pod web` | Everything Kubernetes knows about the pod. The events at the bottom tell its story. |
| `kubectl logs web --tail=4` | Print the last four lines the container wrote. |
| `kubectl exec web -- nginx -v` | Run a command inside the container: the web server's version. |
| `kubectl exec web -- hostname` | The pod's host name, which is its name. |
| `kubectl run web2 --image=nginx:alpine --dry-run=client -o yaml > pod.yaml` | Print a pod as YAML instead of creating it, and save it to a file. |
| `kubectl apply -f pod.yaml` | Create the pod the file describes. |
| `kubectl get pods` | List pods: one made on the command line, one from YAML. |
| `kubectl delete pod web web2` | Delete both pods. A bare pod isn't recreated when it's deleted. |

## EPISODE 6 Kubernetes labels and selectors

| | |
|---|---|
| `kubectl create deployment web --image=nginx:alpine --replicas=3` | Run three nginx pods. The deployment labels each one app=web. |
| `kubectl get pods --show-labels` | List pods with all their labels. |
| `kubectl get pods -l app=web` | Select pods by label: only the app=web pods. |
| `kubectl get pods -l app=api` | A selector that matches no pod returns nothing. |
| `kubectl label pod POD_NAME tier=frontend` | Add a label to one pod. Use a pod name from the list. |
| `kubectl get pods -l tier=frontend` | Select by the new label: just that pod. |
| `kubectl label pod POD_NAME app=debug --overwrite` | Change a pod's app label: the deployment stops counting it and starts a replacement. |
| `kubectl get pods -L app` | Show the app label as a column for every pod. |
| `kubectl delete deployment web` | Delete the deployment: it removes only the pods it still selects. |
| `kubectl delete pod POD_NAME` | Delete the leftover pod by its name. |

## EPISODE 7 Rolling updates and rollbacks

| | |
|---|---|
| `kubectl create deployment web --image=nginx:1.30-alpine --replicas=4` | Run four pods of nginx 1.30. |
| `kubectl annotate deployment web kubernetes.io/change-cause="nginx 1.30"` | Record why this revision exists. The rollout history shows it. |
| `kubectl rollout status deployment/web` | Wait until every pod of the rollout is up. |
| `kubectl set image deployment/web nginx=nginx:1.31-alpine` | Move to a newer image: Kubernetes replaces the pods a few at a time. |
| `kubectl get rs` | List the ReplicaSets: one per version. The old one is scaled to zero but kept. |
| `kubectl rollout history deployment/web` | List the deployment's revisions, each with its change cause. |
| `kubectl set image deployment/web nginx=nginx:1.99-alpine` | The mistake: a version that doesn't exist. |
| `kubectl rollout status deployment/web --timeout=20s` | The new pods never get ready, so this gives up after 20 seconds. |
| `kubectl get pods` | Why: the new pods can't pull their image, while three old pods keep serving. |
| `kubectl rollout undo deployment/web --to-revision=2` | The fix: roll back to revision 2, the last good version. |
| `kubectl get deployment web -o wide` | Confirm which image the deployment runs now. |
| `kubectl delete deployment web` | Remove the deployment, with its ReplicaSets and pods. |

## EPISODE 8 Kubernetes services explained

| | |
|---|---|
| `kubectl create deployment web --image=traefik/whoami --replicas=3` | Run three copies of a small server that replies with its own host name. |
| `kubectl expose deployment web --port=80` | Put a Service, with one stable address, in front of the pods. |
| `kubectl get svc web -o wide` | The Service's address and the selector it uses to find its pods. |
| `kubectl run client --image=curlimages/curl --command -- sleep 3600` | Start a client pod to make requests from inside the cluster. |
| `kubectl exec client -- curl -s web` | Call the Service by its short name. |
| `kubectl exec client -- nslookup web.default.svc.cluster.local` | Look up the Service's full DNS name. |
| `for i in 1 2 3 4 5 6; do kubectl exec client -- curl -s web \| grep Hostname; done` | Six requests in a row: the replies come from different pods. |
| `kubectl get endpointslices -l kubernetes.io/service-name=web` | The pod addresses the Service sends traffic to. |
| `kubectl get pods -l app=web -o wide` | The same addresses, seen from the pods. |
| `kubectl expose deployment web --name=web2 --port=80 --selector=app=webb` | The mistake: a Service whose selector has a typo. |
| `kubectl exec client -- curl -sS -m 3 web2` | The request fails: that Service has no pods behind it. |
| `kubectl delete deployment web && kubectl delete service web web2 && kubectl delete pod client --now` | Remove the deployment, both Services and the client. |

## EPISODE 9 port-forward vs NodePort

| | |
|---|---|
| `kubectl create deployment web --image=traefik/whoami --replicas=3` | Run three copies of the server that replies with its host name. |
| `kubectl expose deployment web --port=80` | Put a ClusterIP Service in front of them. |
| `kubectl port-forward svc/web 8080:80 & sleep 2` | Forward local port 8080 to the Service, in the background. |
| `for i in 1 2 3 4; do curl -s localhost:8080 \| grep Hostname; done` | Four requests through the forward: they all reach the same pod. |
| `fg (then Ctrl+C stops the port-forward)` | Bring the forward back to the foreground, then stop it with Ctrl+C. |
| `curl -sS localhost:8080` | With the forward stopped, nothing listens on 8080 anymore. |
| `kubectl apply -f - <<EOF (the NodePort YAML on screen)` | Create a NodePort Service on port 30080. |
| `kubectl get svc web-np` | The Service's ports: 80 inside the cluster, 30080 on every node. |
| `kubectl describe nodes \| grep InternalIP` | Find the node's address. |
| `for i in 1 2 3 4 5 6; do curl -s <node-ip>:30080 \| grep Hostname; done` | Six requests to the node's port: the replies come from different pods. |
| `kubectl create service nodeport web-bad --tcp=80:80 --node-port=8080` | The mistake: a node port outside 30000 to 32767 is refused. |
| `kubectl delete deployment web && kubectl delete service web web-np` | Remove the deployment and both Services. |

## EPISODE 10 Kubernetes namespaces

| | |
|---|---|
| `kubectl get namespaces` | List the namespaces a new cluster starts with. |
| `kubectl get pods -n kube-system` | -n picks a namespace: here, the pods that run Kubernetes itself. |
| `kubectl create namespace staging` | Create a namespace. |
| `kubectl create deployment hello --image=traefik/whoami -n staging` | Run a deployment in that namespace. |
| `kubectl get pods -n staging` | List the pods in staging. |
| `kubectl get pods` | The mistake: without -n, kubectl looks in the default namespace and finds nothing. |
| `kubectl get pods -A` | List pods in every namespace. |
| `kubectl config set-context --current --namespace=staging` | Make staging the default namespace for the context you're using. |
| `kubectl expose deployment hello --port=80` | Expose hello as a Service. It goes into staging, the current default. |
| `kubectl run client -n default --image=curlimages/curl --command -- sleep 3600` | Start a client pod in the default namespace. |
| `kubectl exec -n default client -- curl -s hello.staging \| grep Hostname` | Call a Service in another namespace by its name, a dot, then its namespace. |
| `kubectl exec -n default client -- curl -sS -m 3 hello` | The short name only works inside the same namespace: this fails. |
| `kubectl delete namespace staging` | Delete the namespace and everything in it. There's no undo. |
| `kubectl config set-context --current --namespace=default` | Switch kubectl back to the default namespace. |
| `kubectl delete pod client --now` | Remove the client pod. |

## EPISODE 11 Kubernetes ConfigMaps

| | |
|---|---|
| `kubectl create configmap app-config \ --from-literal=LOG_LEVEL=info --from-literal=GREETING=hello` | Create a ConfigMap with two settings. |
| `kubectl get configmap app-config -o yaml` | Show it as YAML: the settings sit under data. |
| `kubectl apply -f - <<EOF (the Deployment YAML on screen)` | Create a deployment that reads the ConfigMap two ways: as environment variables and as files. |
| `kubectl rollout status deployment/web` | Wait until its pod is running. |
| `kubectl exec deploy/web -- printenv LOG_LEVEL` | Read the setting as an environment variable. |
| `kubectl exec deploy/web -- ls /etc/config` | The mounted folder holds one file per key. |
| `kubectl exec deploy/web -- cat /etc/config/LOG_LEVEL; echo` | Read the setting as a file. |
| `kubectl patch configmap app-config -p '{"data": {"LOG_LEVEL":"debug"}}'` | Change one value in the ConfigMap. |
| `kubectl exec deploy/web -- sh -c 'while sleep 5; do cat /etc/config/LOG_LEVEL; echo; done'` | Print the mounted file every five seconds: it changes with no restart. Ctrl+C stops the loop. |
| `kubectl rollout restart deployment/web` | The fix for environment variables: restart the pods, so they read the new value. |

## EPISODE 12 Kubernetes secrets

| | |
|---|---|
| `kubectl create secret generic db-pass --from-literal=password=S3cret-demo` | Create a Secret from a value you type. |
| `kubectl get secret db-pass -o yaml` | Show it as YAML: the value is stored base64-encoded, under data. |
| `kubectl get secret db-pass -o jsonpath='{.data.password}' \| base64 -d; echo` | Decode it in one command. Base64 is an encoding, not encryption. |
| `echo -n S3cret-demo \| base64` | Encode the plain text yourself: the same string comes back. |
| `kubectl apply -f - <<EOF (the Pod YAML on screen)` | Create a pod that mounts the Secret as read-only files. |
| `kubectl wait --for=condition=Ready pod/app --timeout=120s` | Wait until the pod is ready. |
| `kubectl exec app -- cat /etc/db/password; echo` | Inside the pod, the file holds the decoded value. |
| `kubectl exec app -- mount \| grep /etc/db` | The mount is tmpfs, in memory, and read-only. |
| `kubectl exec app -- touch /etc/db/test` | Writing there fails: read-only file system. |
| `kubectl auth can-i get secrets` | Ask the API server whether you may read Secrets: yes, as the cluster admin. |
| `kubectl auth can-i get secrets --as=system:serviceaccount:default:default` | The same question as the default service account that pods run as: no. |
| `kubectl apply -f - <<EOF (Secret YAML: data fails, stringData works)` | A plain value under data is rejected. Under stringData it's accepted, and encoded for you. |
| `kubectl get secret api-key -o jsonpath='{.data.key}'; echo` | The stringData value, stored base64-encoded like every Secret value. |

## EPISODE 13 kubectl apply, diff, explain

| | |
|---|---|
| `kubectl create deployment web --image=nginx:alpine \ --dry-run=client -o yaml > web.yaml` | Write a deployment's YAML to a file instead of creating it. |
| `cat web.yaml` | Read the file. |
| `kubectl apply -f web.yaml` | Create what the file describes. Run it again and it says unchanged. |
| `kubectl explain deployment.spec.replicas` | Look up what any field means, from the cluster itself. |
| `sed -i 's/replicas: 1/replicas: 3/' web.yaml` | Edit the file: three replicas. |
| `kubectl diff -f web.yaml` | Preview what apply would change, before you apply it. |
| `kubectl scale deployment web --replicas=5` | The mistake: a quick fix made by hand on the live object. |
| `kubectl get deployment web` | After a plain apply, the hand fix is gone without a word. |
| `kubectl apply --server-side -f web.yaml` | Server-side apply records who manages each field, and refuses to overwrite a hand change. |
| `kubectl apply --server-side --force-conflicts -f web.yaml` | Overwrite it on purpose: the file's value wins. |

## EPISODE 14 Kubernetes probes

| | |
|---|---|
| `kubectl apply -f - <<EOF (the Deployment YAML on screen)` | Create a deployment with a readiness probe and a liveness probe. |
| `kubectl expose deployment web --port=80` | Put a Service in front of it. |
| `kubectl rollout status deployment/web` | Wait until its pods are up. |
| `kubectl describe endpointslices -l kubernetes.io/service-name=web \| grep -E 'Addresses\|Ready'` | Which pod addresses the Service sends traffic to, and whether each one is ready. |
| `P=$(kubectl get pods -l app=web -o name \| head -1)` | Keep one pod's name in $P. |
| `kubectl exec $P -- rm /usr/share/nginx/html/index.html` | Break readiness: remove the page the readiness probe checks. |
| `kubectl get pods -w (Ctrl+C stops it)` | Watch that pod go to 0/1: still running, but taken out of the Service. Ctrl+C stops the watch. |
| `kubectl exec $P -- sh -c 'echo ok \| tee /usr/share/nginx/html/index.html'` | Put the page back: the pod is ready again. |
| `kubectl exec $P -- rm /tmp/alive` | Break liveness: remove the file the liveness probe checks. |
| `kubectl events --for $P \| grep -E 'Liveness\|restarted'` | The failed liveness checks, and the restart they caused. |
| `kubectl apply -f - <<EOF (the bad Pod: liveness on port 8080)` | The mistake: a liveness probe pointed at the wrong port. |
| `kubectl get pod bad -w` | Watch the pod restart until it reaches CrashLoopBackOff. |
| `kubectl events --for pod/bad \| grep Liveness \| tail -1` | Why: the probe's connection to port 8080 is refused. |

## EPISODE 15 Requests and limits

| | |
|---|---|
| `cat > hog.yaml <<EOF (the hog Pod on screen: limit 100Mi)` | Write a pod that uses more memory than its 100Mi limit. |
| `kubectl apply -f hog.yaml` | Create it. |
| `kubectl get pod hog -w (Ctrl+C stops it)` | Watch it get OOMKilled and restarted. Ctrl+C stops the watch. |
| `kubectl describe pod hog \| grep -A 2 'Last State'` | Why it stopped: OOMKilled, exit code 137. |
| `kubectl delete pod hog --now` | Remove it. |
| `cat > big.yaml <<EOF (the big Pod on screen: requests 100Gi)` | The mistake: a pod that requests 100Gi of memory. |
| `kubectl apply -f big.yaml` | Create it. |
| `kubectl get pod big` | It stays Pending: no node has that much memory. |
| `kubectl events --for pod/big \| tail -1` | The scheduler's reason: not enough memory. |
| `sed -i 's/100Gi/100Mi/' big.yaml` | The fix: request what the app really needs. |
| `kubectl delete pod big --now` | Remove the Pending pod. A pod's requests can't be changed in place. |
| `kubectl apply -f big.yaml` | Create it again. |
| `kubectl wait --for=condition=Ready pod/big --timeout=90s` | Now it's scheduled and becomes ready. |

Kubernetes Hands-On is a free, screen-recorded series from AIS: every command here was typed live on a real cluster. POD_NAME stands for a name from your own kubectl get pods output, and <node-ip> for your node's InternalIP. Kubernetes is a registered trademark of The Linux Foundation; this series is independent and not affiliated with or endorsed by it.