Omar Latrach

TECHNICAL ARTICLE

PUBLISHED · SEPTEMBER 17, 2026

DaemonSets, StatefulSets, Jobs, and CronJobs: Choosing the Right Workload Controller

Choose Kubernetes workload controllers by what they preserve: node coverage, member identity, completed work, or a schedule. Follow each lifecycle with YAML and kubectl.

This is Learn post 08 of the 54-post Certified Kubernetes Administrator (CKA) preparation path. Posts 06 and 07 introduced Deployments, ReplicaSets, and controlled Pod replacement. Now we keep that reconciliation model and change the condition the controller must maintain.

Three web replicas, one agent on every eligible node, two named database members, and a nightly report are different administrative intentions. Choosing a controller means choosing which intention Kubernetes will keep trying to satisfy.

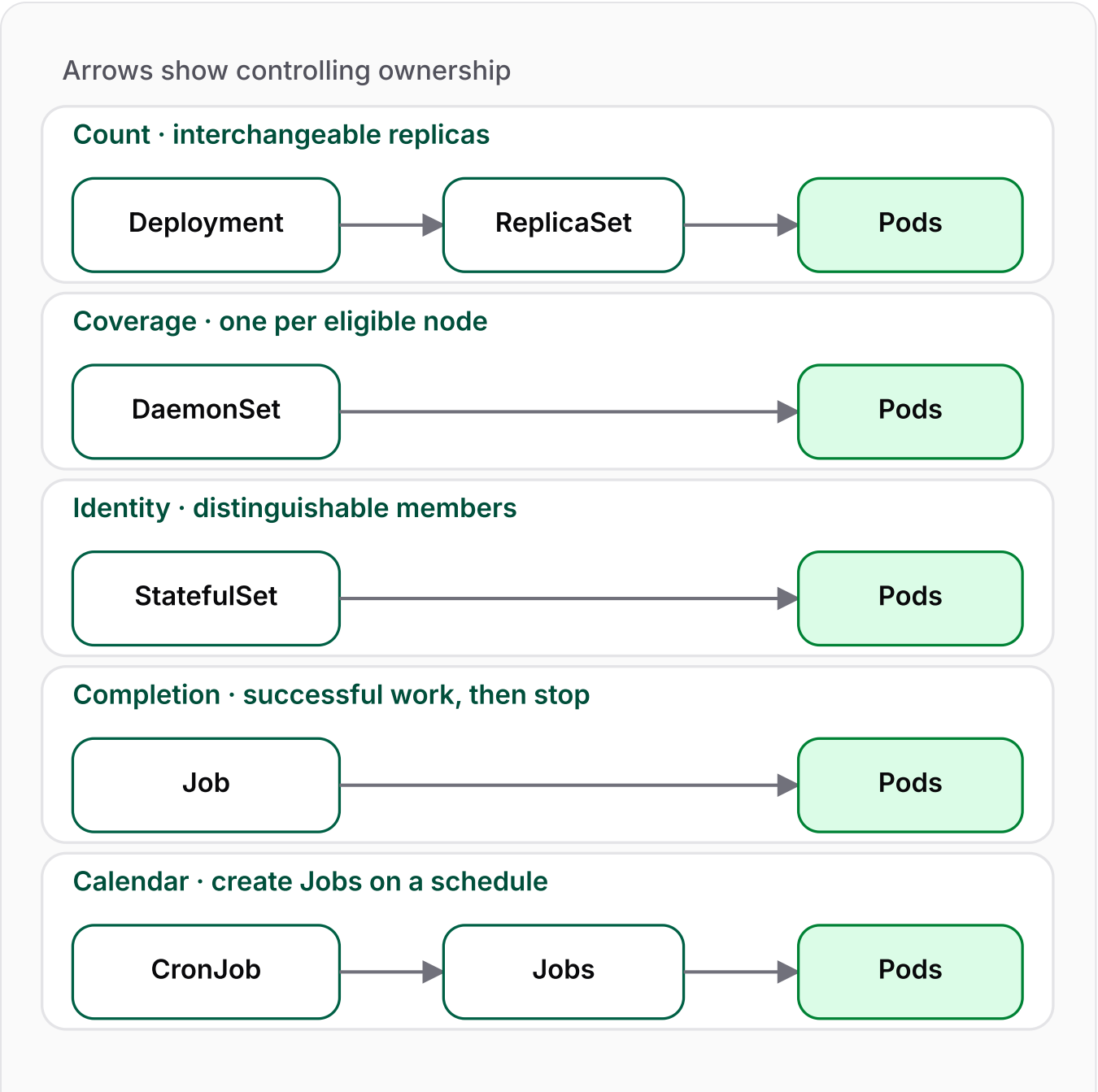
What you'll learn

- Choose a controller by the condition it preserves: replica count, node coverage, member identity, completed work, or schedule.
- Read and apply compact DaemonSet, StatefulSet, Job, and CronJob manifests.
- Follow ownership, readiness, completion counts, and scheduled Job creation with kubectl.
- Distinguish Pod replacement from container retries, durable storage from stable names, and missed starts from runtime deadlines.

The mental model: what must remain true?

> Count → Deployment. Coverage → DaemonSet. Identity → StatefulSet. Completion → Job. Calendar → CronJob.

- Deployment: keep N interchangeable application replicas running. Pod identity is disposable.
- DaemonSet: keep a Pod on each eligible node. The node set determines the target, rather than spec.replicas.
- StatefulSet: keep N distinguishable members, each with a stable identity and, when configured, its own persistent storage.
- Job: reach a successful completion target, then stop creating work.
- CronJob: create Jobs when a schedule calls for them. Each Job handles its own Pods and completion.



Choose the controller by the condition it maintains

Count, coverage, identity, completion, or calendar. Arrows point from the controlling owner to its dependent resources; controllers act through the Kubernetes API.

All four new controllers use a Pod template. DaemonSets, StatefulSets, and Jobs manage Pods directly; there is no ReplicaSet between them and their Pods. A CronJob has a Job template containing another Pod template. The controller requests objects through the API; the scheduler and kubelet handle placement and execution.

A small, normal demonstration environment

The following demonstrations share an isolated namespace and need a working cluster, permission to create these resources, Linux nodes, and access to the example container images. Save each full YAML block under its displayed filename. The StatefulSet demonstration needs no storage provisioner because it teaches identity first.

```bash
kubectl create namespace workload-demo
```

Commands run in this namespace throughout. Read their resulting objects as evidence of the controller contract, rather than expecting one fixed sequence of timings or generated names.

DaemonSet: coverage follows the nodes

Node logging agents and monitoring agents normally need a local instance on every relevant node. A Deployment with three replicas could place two on one node and none on another. A DaemonSet expresses the missing requirement: coverage.

Because its target is the eligible node set, a DaemonSet creates a Pod for a newly eligible node and replaces a lost managed Pod on a node that still qualifies. There is no replicas field to scale. During updates, transient states can differ from the steady-state one-Pod-per-node model.

Declare a harmless node agent

This agent prints a heartbeat; it does not collect real host metrics. The Pod selector and template labels match, just as in the earlier Deployment lesson. The one nodeSelector entry limits this Linux container to Linux nodes; detailed placement rules belong to post 11.

node-agent.yaml · yaml

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-agent
  namespace: workload-demo
spec:
  selector:
    matchLabels:
      app: node-agent
  template:
    metadata:
      labels:
        app: node-agent
    spec:
      nodeSelector:
        kubernetes.io/os: linux
      containers:
        - name: agent
          image: busybox:1.37.0
          command: ["sh", "-c"]
          args:
            - 'while true; do echo "node agent alive"; sleep 30; done'
```

Observe coverage rather than a replica target

Apply the declaration, wait for rollout completion, and compare the DaemonSet counts with the actual node placement. Use describe when the aggregate counts need explanation.

bash

```bash
kubectl apply -f node-agent.yaml
kubectl rollout status daemonset/node-agent -n workload-demo --timeout=2m
kubectl get daemonset/node-agent -n workload-demo
kubectl get pods -n workload-demo -l app=node-agent -o wide
kubectl describe daemonset/node-agent -n workload-demo
```

In the DaemonSet row, DESIRED is the number of nodes that should run a managed Pod. CURRENT is the number of nodes running one; READY is the number with a ready managed Pod. On a healthy two-node eligible set, all three reach two. The Pods’ NODE column shows the distribution that a replica count alone would not guarantee.

Eligible does not mean every machine unconditionally. Node selection and taints affect coverage. A taint is a node restriction, and a toleration allows a Pod to tolerate one. DaemonSet Pods receive some built-in tolerations, but this example does not tolerate the usual control-plane NoSchedule taint. A cluster whose only node has that taint may show no coverage. Post 12 teaches those rules; do not add a blanket toleration just to change the count.

Change the controller template, then observe replacement

Post 07 established that running Pods do not change image in place. DaemonSets also support template-driven replacement. Their default RollingUpdate strategy replaces managed Pods under an availability limit; OnDelete instead waits for an administrator to delete old Pods before replacements use the new template.

For an operational restart under the default strategy, use the controller and wait for its rollout. This patches the template and replaces Pods; it does not restart a container inside its existing Pod.

```bash
kubectl rollout restart daemonset/node-agent -n workload-demo
kubectl rollout status daemonset/node-agent -n workload-demo --timeout=2m
```

Official DaemonSet documentation explains node coverage and controller behavior.

DaemonSet rolling-update guide documents its replacement strategies.

StatefulSet: preserve the member, replace the Pod

Some applications distinguish member zero from member one. A database cluster might associate each member with a network name and its own disk. If a failed member came back as a random new replica, the application could lose that association. A StatefulSet gives replacements the same logical member identity.

The default ordinal sequence starts at zero: members-0, members-1, and so on. Replacement preserves the Pod name, not the original Pod object: its unique identifier (UID) changes. IP addresses and in-memory data are not preserved by that name.

See identity without introducing storage administration

The governing headless Service below supplies the network domain for these member names. Headless means clusterIP: None: there is no single virtual Service IP. Create the Service yourself; the StatefulSet does not create it. Service routing and Domain Name System (DNS) details belong to posts 28 and 29.

This NGINX example is intentionally not a replicated database and has no persistent volumes. Its readiness probe, introduced in post 05, gives the controller an explicit ready signal before it proceeds to the next member.

members.yaml · yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: members
  namespace: workload-demo
spec:
  clusterIP: None
  selector:
    app: members
  ports:
    - name: http
      port: 80
      targetPort: http
---
apiVersion: apps/v1
```

```
kind: StatefulSet
metadata:
  name: members
  namespace: workload-demo
spec:
  serviceName: members
  replicas: 2
  selector:
    matchLabels:
      app: members
  template:
    metadata:
      labels:
        app: members
    spec:
      containers:
        - name: nginx
          image: nginx:1.28.0-alpine
          ports:
            - name: http
              containerPort: 80
          readinessProbe:
            tcpSocket:
              port: http
            periodSeconds: 2
```

Apply both objects, then inspect the members' names and ownership. The custom columns expose the actual managing relationship rather than relying on a name that looks related.

bash

```
kubectl apply -f members.yaml
kubectl rollout status statefulset/members -n workload-demo --timeout=2m
kubectl get statefulset/members -n workload-demo
kubectl get pods -n workload-demo -l app=members   -o custom-columns='NAME:.metadata.name,OWNER-
KIND:.metadata.ownerReferences[0].kind,OWNER:.metadata.ownerReferences[0].name,UID:.metadata.uid'
```

The Pod names are members-0 and members-1, and both controlling owners are StatefulSet/members. Under the default OrderedReady policy, member zero becomes Running and Ready before member one is created. Readiness gates progress; application replication is still the application's responsibility.

Observe a normal member replacement

After the initial rollout completes, record member one's UID, delete that Pod normally, and watch until the replacement members-1 is ready. Stop the watch with Ctrl-C, then retrieve the new UID. Watching the replacement avoids assuming that an asynchronous status update has already occurred.

```bash
kubectl get pod/members-1 -n workload-demo   -o jsonpath='{.metadata.name}{"  "}{.metadata.uid}{"\n"}'
kubectl delete pod/members-1 -n workload-demo
kubectl get pods -n workload-demo -l app=members --watch
# Stop with Ctrl-C after the replacement members-1 shows READY 1/1.
kubectl get pod/members-1 -n workload-demo   -o jsonpath='{.metadata.name}{"  "}{.metadata.uid}{"\n"}'
```

The name remains members-1 while the UID changes. That is the useful distinction: Kubernetes replaced an object while preserving its logical slot. Use normal deletion here; forcing deletion when the old process may still be alive can break the single-member identity assumption.

Order affects scaling and updates

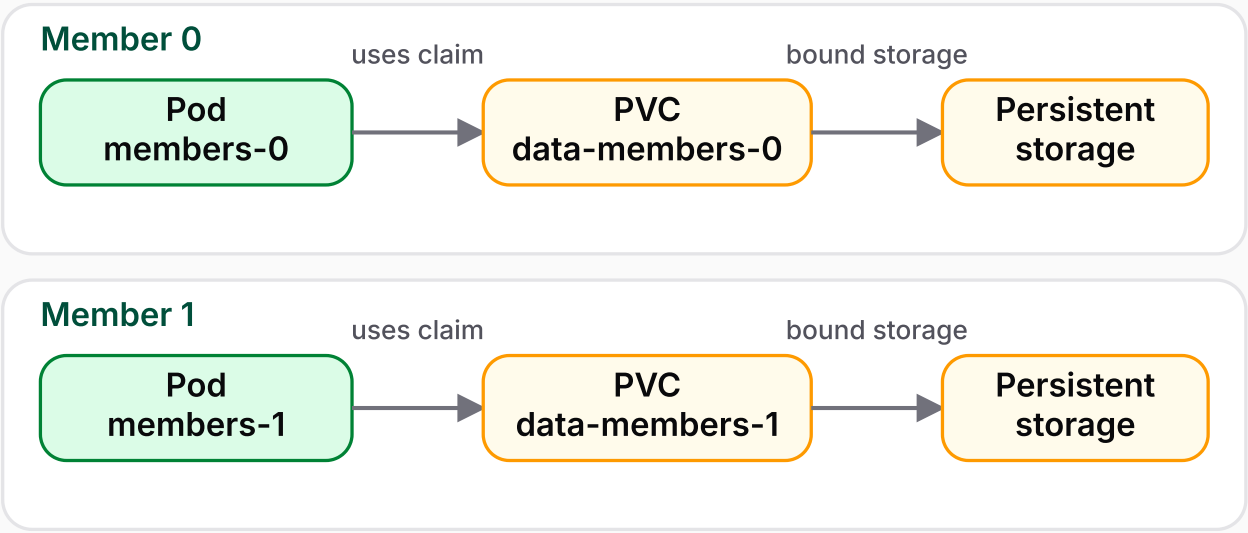
To add a third member, scale the StatefulSet and observe the resulting Pod. Under OrderedReady, Kubernetes adds members-2 after its predecessors are ready. Scaling back down removes the highest ordinal first, so members-2 goes before members-1.

```bash
kubectl scale statefulset/members -n workload-demo --replicas=3
kubectl rollout status statefulset/members -n workload-demo --timeout=2m
kubectl get pods -n workload-demo -l app=members
kubectl scale statefulset/members -n workload-demo --replicas=2
```

The default RollingUpdate strategy updates from the highest ordinal downward and waits for readiness before continuing. StatefulSets also support OnDelete. These are StatefulSet policies, not the Deployment's maxSurge-based exchange between ReplicaSets. Keep members.yaml aligned with the intended count whenever you change it operationally.

Stable storage needs an explicit declaration

A StatefulSet name does not make a container filesystem durable. With volumeClaimTemplates, the controller creates a separate PersistentVolumeClaim (PVC), a request for persistent storage, for each member. A data template on this StatefulSet produces data-members-0 and data-members-1. A replacement member reuses its existing claim.



Configured StatefulSet storage stays associated with each member

With volumeClaimTemplates configured and claims bound, each member uses its own PersistentVolumeClaim (PVC). A replacement reuses that claim. This is a logical storage association, not a network path.

For recognition, the fragment below belongs under a storage-backed StatefulSet's spec. A container must also mount the claim template name with volumeMounts. Do not add this fragment to the identity demonstration: satisfying claims needs suitable persistent storage, and this lesson deliberately does not assume a StorageClass or provisioner.

Recognition fragment only: claim template · yaml

```yaml
spec:
  volumeClaimTemplates:
    - metadata:
        name: data
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: 1Gi
```

By default, scaling down or deleting a StatefulSet retains its claims rather than deleting application data. An explicit PVC retention policy can change that behavior. StatefulSets do not provide database replication, backups, or recovery from damaged data. Volumes, claim binding, access modes, and provisioning belong to posts 33–35.

Official StatefulSet documentation covers identity, ordering, updates, and persistent-storage associations.

Job: count successful work, then stop

A report generator or import process should exit after completing its work. A Deployment's default Pod restart behavior would restart even a successfully exited container; a Job instead treats successful termination as progress toward a completion target.

For this ordinary fixed-completion Job, completions is the number of successful Pods needed and parallelism is the requested maximum number running at once. Three completions with parallelism two means up to two active workers toward three

successes, not two successes per round forever.

Declare a finite, visible task

This demonstration prints one result per successful Pod. Each worker runs the same command; setting completions does not assign three different business records. Real parallel work needs application-level work allocation or another explicit assignment mechanism.

report-job.yaml · yaml

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: report
  namespace: workload-demo
spec:
  completions: 3
  parallelism: 2
  backoffLimit: 2
  activeDeadlineSeconds: 120
  ttlSecondsAfterFinished: 600
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: report
          image: busybox:1.37.0
          command: ["sh", "-c"]
          args:
            - 'sleep 2; echo "report finished"'
```

Use the manifest when the completion and retry settings matter. When only a one-off command is needed, the imperative generator is a useful starting point; its client-side dry run prints YAML without creating a Job. The preview below creates no object and is separate from report-job.yaml.

bash

```bash
kubectl create job report-preview -n workload-demo   --image=busybox:1.37.0 --dry-run=client -o yaml   -- sh -c 'echo "report finished"'
```

Observe completion rather than readiness

Apply the actual manifest and wait for its Complete condition. Inspect the Pods and logs before the ten-minute cleanup window expires. A Job-level log command selects a Pod; use the label selector when you want all workers' output.

```bash
kubectl apply -f report-job.yaml
kubectl wait --for=condition=complete job/report -n workload-demo --timeout=2m
kubectl get job/report -n workload-demo
kubectl get pods -n workload-demo -l batch.kubernetes.io/job-name=report
kubectl logs -n workload-demo   -l batch.kubernetes.io/job-name=report --prefix=true
kubectl get job/report -n workload-demo   -o jsonpath='{.status.succeeded}{" successful Pods\n"}'
```

After success, the Job's completion count reaches 3/3 and the last command prints "3 successful Pods". The Pod list normally displays Completed for each worker; the Pod API phase is Succeeded. Logs contain "report finished" from each successful worker. These terminated Pods are evidence of finished work, not running capacity the Job must replace.

Retries, deadlines, and cleanup answer different questions

- restartPolicy: Never means the kubelet does not restart the failed container inside that Pod. The Job controller can create a replacement Pod while retries remain.
- restartPolicy: OnFailure is the other valid Job Pod policy. It allows container restarts within the same Pod; those retries also contribute to the Job's backoff accounting. Always is not valid for a Job Pod template.
- backoffLimit: 2 bounds retry behavior after failure; it is not a parallelism setting. The controller backs off retries rather than creating replacements continuously.
- activeDeadlineSeconds: 120 bounds the total active Job duration, including retry delays. It can terminate remaining work before the retry budget is used up; it is not a fresh 120 seconds for every Pod.

- ttlSecondsAfterFinished: 600 makes a finished Job eligible for automatic deletion after ten minutes, whether it succeeded or failed. Deleting the Job also cleans up its dependent Pods; collect needed evidence first.

kubectl wait --timeout only limits how long your client waits. It does not set a Job runtime deadline, and a timeout does not stop the workload. If the Complete condition never appears, inspect the Job's conditions and counts with describe; a Failed Job is terminal and does not restart itself indefinitely.

```bash
kubectl describe job/report -n workload-demo
```

A successful completion count is not an exactly-once transaction guarantee. Retries or disruptions can run a program more than once. Design side effects to be idempotent: repeating the same operation should not create an unwanted duplicate result.

Most ordinary Job Pod-template changes are immutable. To run different work after completion, create a new Job with a new name instead of expecting a finished Job to roll out a changed image. Some specialized update exceptions exist, but they are unnecessary for this controller mental model.

Official Job documentation explains completion tracking and retry behavior.

Job API reference defines the deadline and cleanup fields used here.

CronJob: a calendar creates Jobs

A CronJob separates "when to start" from "how to finish". It creates a new Job from jobTemplate for a scheduled occurrence; that Job creates Pods. An existing completed Job is not restarted each night.

Declare the schedule outside the Job template

This harmless report is scheduled every minute so its normal lifecycle is visible. The five schedule fields are minute, hour, day of month, month, and day of week. For a daily 02:00 report, the expression would be "0 2 * * *". The quoted string keeps the YAML unambiguous.

scheduled-report.yaml · yaml

```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: scheduled-report
  namespace: workload-demo
spec:
  schedule: "* * * * *"
  timeZone: "Etc/UTC"
  concurrencyPolicy: Forbid
  startingDeadlineSeconds: 120
  successfulJobsHistoryLimit: 2
  failedJobsHistoryLimit: 1
  jobTemplate:
    spec:
      backoffLimit: 2
      activeDeadlineSeconds: 45
      template:
        spec:
          restartPolicy: Never
          containers:
            - name: report
              image: busybox:1.37.0
              command: ["sh", "-c"]
              args:
                - 'date; sleep 5; echo "scheduled report finished"'
```

timeZone is stable in Kubernetes 1.27 and later. Etc/UTC means Coordinated Universal Time (UTC), independent of the administrator’s laptop. Without timeZone, the controller uses its own local time zone; do not embed TZ or CRON_TZ inside schedule.

Notice the nesting: schedule and overlap policy belong to the CronJob, retry and runtime settings belong under jobTemplate.spec, and restartPolicy belongs under jobTemplate.spec.template.spec. The two deadline fields regulate different stages.

Follow one scheduled occurrence through ownership

Apply the schedule and watch Jobs appear. Creation waits for a scheduled occurrence; it is not an immediate consequence of apply. Stop the watch with Ctrl-C after observing an occurrence, then inspect the owning CronJob and the Jobs' completion counts.

```bash
kubectl apply -f scheduled-report.yaml
kubectl get cronjob/scheduled-report -n workload-demo
kubectl get jobs -n workload-demo --watch
# After observing an occurrence, stop the watch with Ctrl-C.
kubectl get jobs -n workload-demo   -o custom-columns='NAME:.metadata.name,OWNER-KIND:.metadata.ownerReferences[0].kind,OWNER:.metadata.ownerReferences[0].name,SUCCEEDED:.status.succeeded'
kubectl get pods -n workload-demo -l batch.kubernetes.io/job-name
```

Scheduled Job names contain generated suffixes; discover them rather than hard-coding one. Their controlling owner is CronJob/scheduled-report. The earlier report Job has no CronJob owner. Each scheduled Job owns its own Pods; a finished worker does not imply that the recurring schedule has stopped.

Control overlap and late starts deliberately

- Allow, the default concurrencyPolicy, allows overlapping Jobs from this CronJob.
- Forbid avoids creating another Job while a previous Job from this CronJob is still active. An overlapping occurrence is missed; it is not a durable queue of every requested run.
- Replace replaces the currently running Job from this CronJob with a new one when the next occurrence arrives. Interruption must be acceptable to the application.
- startingDeadlineSeconds: 120 sets the allowed lateness for creating a Job for a scheduled occurrence. It does not limit how long that Job runs. Here activeDeadlineSeconds: 45 inside the Job template controls runtime instead.
- The history limits retain up to two successful scheduled Jobs and one failed scheduled Job. They are counts of finished Job records, not runtime deadlines or retries.

Concurrency policy applies only to Jobs created by the same CronJob. It does not prevent another CronJob or an independently created Job from performing the same work. Schedules are approximate: circumstances can cause missed or duplicate starts. The application still needs safe handling of repeated execution.

Pause future starts or create one immediate run

Suspend a schedule for an operational pause, such as a maintenance window. Setting suspend does not cancel Jobs that are already running. Resume when future scheduling is wanted again; on resume, missed occurrences may qualify for creation within the starting deadline.

```bash
kubectl patch cronjob/scheduled-report -n workload-demo   --type=merge -p '{"spec":{"suspend":true}}'
kubectl get cronjob/scheduled-report -n workload-demo
kubectl patch cronjob/scheduled-report -n workload-demo   --type=merge -p '{"spec":{"suspend":false}}'
```

For an immediate operational run using the same Job template, create a separately named Job from the CronJob. It is independent of scheduled occurrence tracking and the CronJob's overlap policy, so account for any scheduled work already running. Use a new name for each later manual run.

```bash
kubectl create job report-manual -n workload-demo   --from=cronjob/scheduled-report
kubectl wait --for=condition=complete job/report-manual   -n workload-demo --timeout=2m
kubectl logs job/report-manual -n workload-demo
```

Editing jobTemplate affects future Jobs only. Existing Jobs and Pods keep the configuration they were created with. As with scaling, keep the declarative source aligned after operational patches.

Official CronJob documentation covers schedules, time zones, suspension, and overlap policies.

kubectl create job reference documents both imperative generation and creating a Job from a CronJob.

Choose by the invariant, then inspect the right evidence

- An HTTP application with interchangeable replicas: use a Deployment; inspect desired, updated, ready, and available replica counts.
- A monitoring agent that belongs on every relevant node: use a DaemonSet; inspect coverage counts and the Pods' NODE column.
- Distinct application members that must keep their identities: use a StatefulSet; inspect ordinals, readiness, and any configured per-member claims.
- One finite import or report: use a Job; inspect successful completions, failure conditions, and worker logs.
- The same finite work on a calendar: use a CronJob; inspect schedule and suspension first, then follow its Jobs and their Pods.

A Pod that is not ready still exists; it does not automatically count as a missing instance for every controller. A finished Job Pod, by contrast, may be exactly the success its controller wanted. Always interpret Pod state through its owner's contract.

Common mistakes to avoid

- Using a Deployment replica count as a guarantee of one Pod per node. Coverage needs a DaemonSet contract.
- Treating a stable StatefulSet Pod name as persistence. Durable data needs configured storage, and application replication needs application support.

- Reading Job parallelism as the completion target. Concurrency and required successes are separate settings.
- Putting a Job's restartPolicy or a CronJob's runtime deadline at the wrong nesting level. Read the template hierarchy before applying.
- Expecting Forbid to serialize every scheduled occurrence eventually. Missed starts have deadline rules, not a guaranteed queue.
- Assuming a controller solves exactly-once execution. Jobs and CronJobs can repeat work; protect application side effects.
- Leaving a live patch out of the manifest. The next apply can restore an older schedule, template, or replica count.

Clean up and connect the next lessons

The namespace contains only these demonstrations. Remove it when finished; this deletes the example controllers, their dependent Jobs and Pods, and the governing Service. This identity-only StatefulSet has no persistent claims to preserve.

```bash
kubectl delete namespace workload-demo
```

Post 07: Rolling Updates, Rollbacks, and Deployment Strategies supplies the earlier template-replacement model. Post 09 next introduces ConfigMaps and Secrets for configuring these workloads. Scheduling details wait for posts 11–13, Services and DNS for posts 28–29, and persistent storage for posts 33–35.

What to remember

- Pick the condition Kubernetes should maintain: count, coverage, identity, completion, or calendar.
- DaemonSet coverage follows eligible nodes; StatefulSet replacement preserves logical member identity, not the original Pod UID or unconfigured data.

- Jobs count successful work. Parallelism controls concurrency; restart policy and backoff govern retry behavior.
- A CronJob creates Jobs, and Jobs create Pods. Schedule edits affect future Jobs; suspension stops future starts.
- A missed-start deadline, a Job runtime deadline, and your kubectl wait timeout govern three different clocks.
- Follow owner references and controller status to understand why a particular Pod exists—or why it should finish and stay finished.