Omar Latrach

TECHNICAL ARTICLE

PUBLISHED · SEPTEMBER 7, 2026

Pods: Lifecycle, Init Containers, and Multi-Container Pods

Understand Pod identity, container restarts, ordered initialization, and shared resources through clear Kubernetes commands and multi-container demonstrations.

This is Learn post 04 of the 54-post Certified Kubernetes Administrator (CKA) preparation path. Posts 01–03 established the API request flow, kubectl targeting, and the difference between metadata, spec, and status. Now we will connect those objects to the containers actually running on a node.

A container can exit successfully, restart repeatedly, or wait for initialization while its Pod still exists. Reading those situations correctly starts with one distinction: **the Pod has an identity and lifetime; each container inside it has its own execution history.**

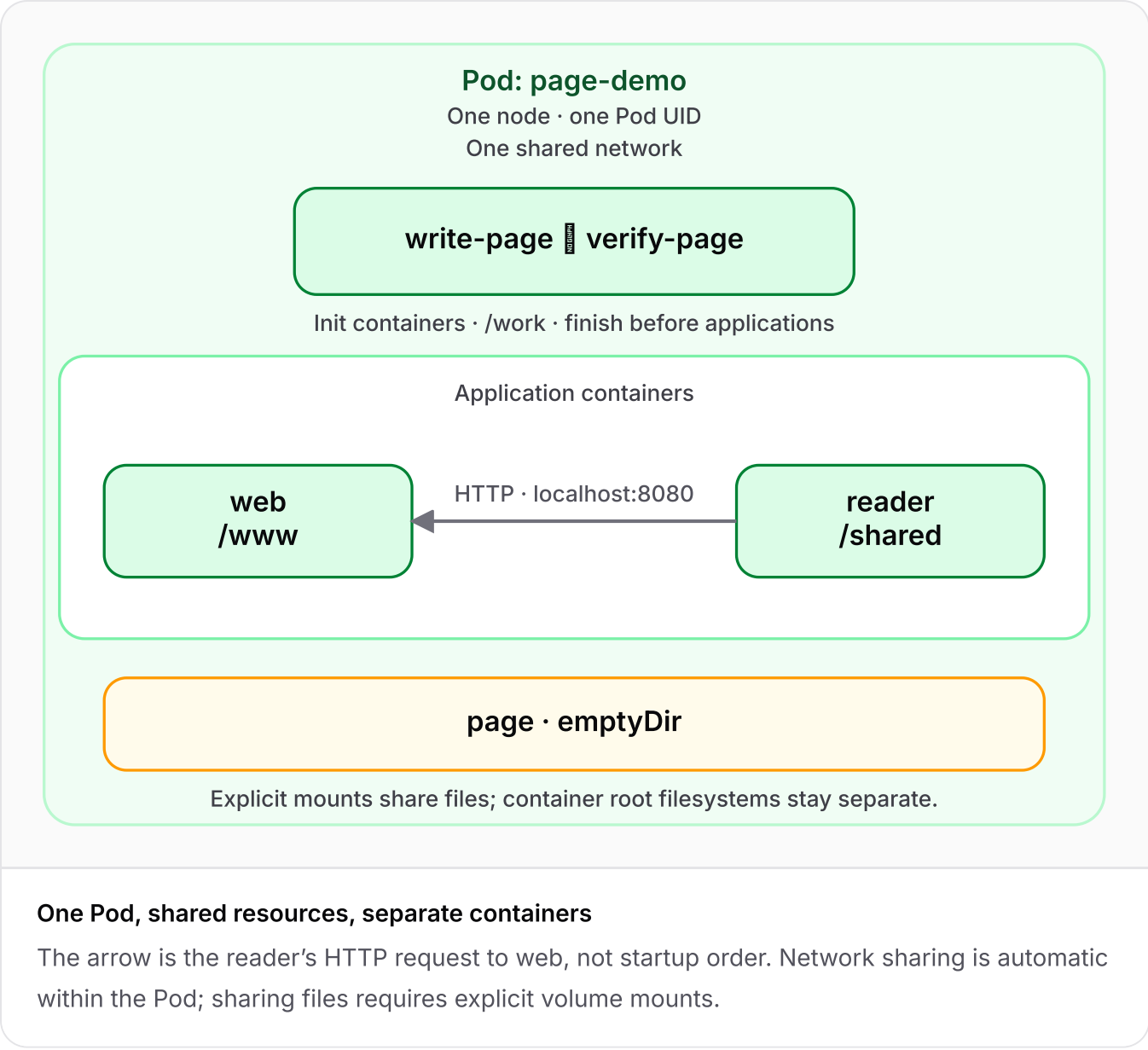
What you'll learn

- Explain what containers share inside a Pod and what remains separate.
- Read Pod phase, container state, and readiness without confusing them.
- Choose a restart policy and recognize container restart versus Pod replacement.
- Use init containers to prepare something before application containers start.
- Observe a multi-container Pod through its files, localhost connection, and container-specific logs.
- Recognize native sidecars and understand normal Pod shutdown.

One home, several processes

Think of a Pod as a temporary home for closely related containers. Kubernetes schedules that home onto one node. Its containers share the Pod network, so they can communicate through `localhost` and must coordinate listening ports. Two servers cannot both bind the same address and port there. Adding another container does not create another Pod IP address.

Each container still has its own image and root filesystem. A file written into one container's ordinary filesystem does not appear in another. To exchange files, the Pod defines a volume and each participating container explicitly mounts it. Process namespaces are also separate by default; sharing the Pod network does not automatically make every container's processes visible to the others. See the official Pods overview and process namespace explanation.



One Pod, shared resources, separate containers

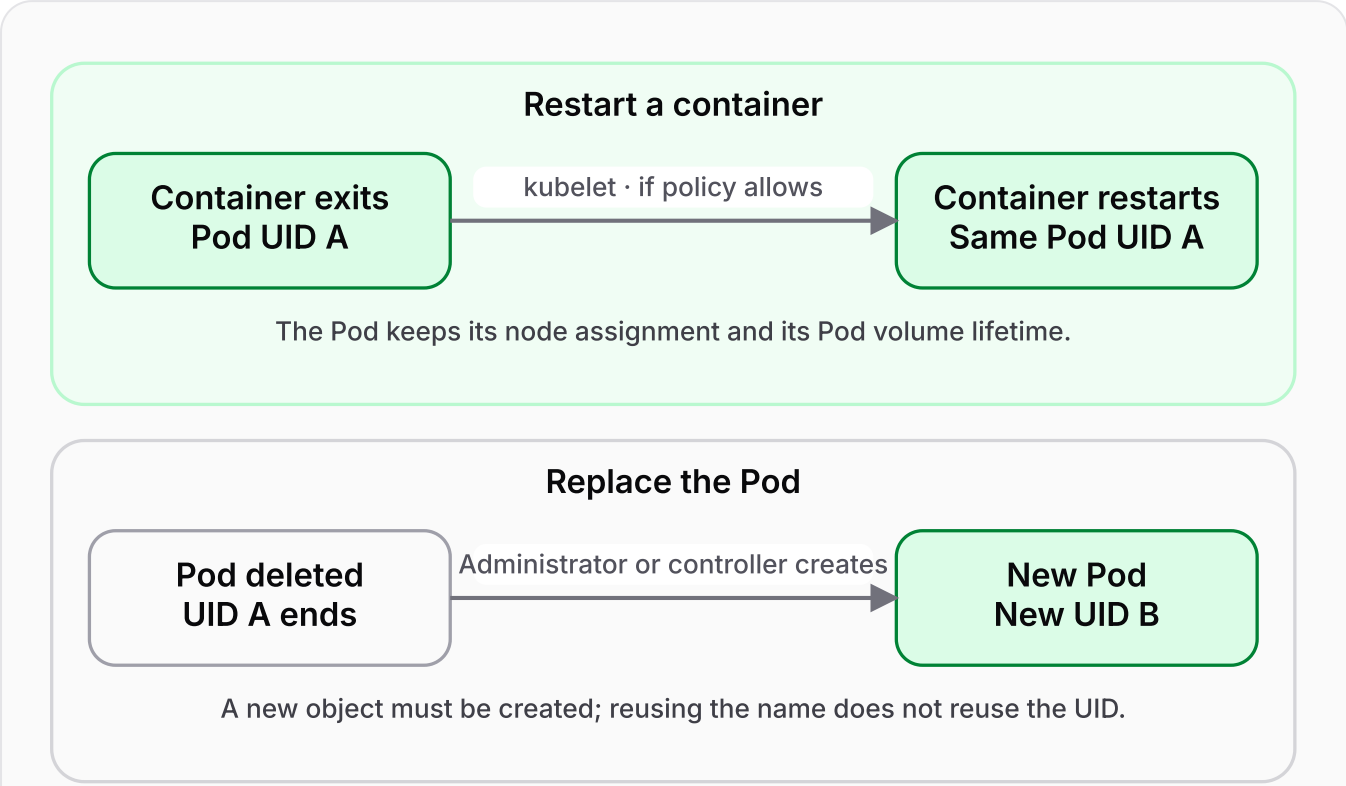
The arrow is the reader's HTTP request to web, not startup order. Network sharing is automatic within the Pod; sharing files requires explicit volume mounts.

Use this grouping when containers form one operational unit: an application and a closely attached helper, for example. An unrelated frontend and database usually need separate Pods because their placement and scaling needs differ. Replication means additional Pods, not more copies stuffed into one Pod; post 06 develops that controller model.

A restart keeps the home; replacement creates a new one

The Pod's `metadata.uid` identifies this particular object. Restarting one container keeps that Pod identity and its node assignment. A replacement Pod has a new UID, even if an administrator reuses the same name. Kubernetes does not move the existing Pod to

another node. A controller can create a replacement; a directly created Pod has no controller to recreate it after deletion.



Container restart and Pod replacement have different identities

A restart preserves the Pod's identity. Replacement requires a separate creation action; deleting a directly created Pod does not automatically create another.

The walkthroughs use direct Pods to make these boundaries visible. For managed application availability, return to this distinction in post 06. The official Pod lifetime documentation describes the identity boundary.

Read three different layers of state

After the API server accepts a Pod, the scheduler selects a node. The kubelet on that node coordinates container startup through the runtime, including preparing the Pod environment, obtaining images, and running initialization. Acceptance therefore precedes execution.

Pod phase is the broad summary in `status.phase`:

- `Pending`: startup is incomplete. This can include waiting for a node, image downloads, or initialization; it does not mean only "unscheduled."
- `Running`: the Pod is assigned to a node, its containers have been created, and at least one is running, starting, or restarting.
- `Succeeded`: all containers finished successfully and will not restart.
- `Failed`: all containers have terminated, at least one failed, and no automatic restart remains.
- `Unknown`: Kubernetes could not obtain the Pod's state, typically because it could not communicate with its node.

Container state is finer grained: `Waiting`, `Running`, or `Terminated`. A terminated container has an exit code and reason; termination can mean successful completion. The Pod API reference documents both Pod and container status fields.

Pod conditions answer specific questions. `PodScheduled` reports placement, `Initialized` reports completion of required initialization, `ContainersReady` reports container readiness, and `Ready` reports Pod readiness. A condition carries a status such as `True` or `False`. This is where the next lesson's health checks will connect; a running process alone does not prove the application can serve requests.

The `STATUS` column from `kubectl get pods` is a display summary, not a literal copy of `status.phase`. `Completed`, `Init:0/2`, `CrashLoopBackOff`, and `Terminating` are examples of display text, not extra Pod phases. Repeated container exits can produce `CrashLoopBackOff` while the Pod phase is still `Running`: the kubelet is delaying another restart. Diagnose the cause in the later troubleshooting lessons; here, remember to read the correct layer. See Pod phase and container states.

A short-lived container can succeed normally

The following walkthroughs assume kubectl access to a learning cluster with Linux worker nodes and permission to create Pods and a namespace. Nodes must be able to pull `busybox:1.37.0`. The native sidecar variation later in the lesson assumes Kubernetes 1.33 or newer, where that feature is stable.

First confirm the target using the workflow from post 02, then create a namespace dedicated to these demonstrations. Use the displayed context to confirm this is your learning cluster before the create command:

```bash
kubectl config current-context
kubectl create namespace pods-learn
```

If `pods-learn` already exists from an earlier run of this lesson, reuse it after checking its contents. Every Pod command below names the namespace explicitly.

Restart policy describes what happens after exit

For the regular application containers in these examples, `spec.restartPolicy` controls whether the kubelet starts another instance after the container exits:

- `Always`, the default: restart after either success or failure.
- `OnFailure`: restart after a nonzero exit code; leave a successful exit alone.
- `Never`: do not restart after either outcome.

These examples do not configure individual container restart overrides. A regular init container also retries on failure under `Always` or `OnFailure`, but a successful init container is allowed to finish. With `Never`, a failed init container fails the Pod. Native sidecars have their own rule, shown later. The fields are described in the Pod specification.

An administrator can use `kubectl run` for a small, direct Pod whose command should finish. Here `--restart=Never` means that printing a message is the entire workload:

```bash
kubectl run greeting -n pods-learn --image=busybox:1.37.0 \
  --restart=Never --command -- sh -c 'printf "Pod work complete\n"'
```

`--command` makes the arguments after `--` the container's command. The image includes `sh`, and `sh -c` executes the following string. Without an explicit shell, Kubernetes does not interpret shell operators such as `>` in a command array. Command and argument configuration explains the relationship to image defaults.

Wait for the outcome this workload is meant to reach, then read its output and phase:

```bash
kubectl wait -n pods-learn pod/greeting \
  --for=jsonpath='{.status.phase}'=Succeeded --timeout=120s
kubectl logs -n pods-learn greeting -c greeting
kubectl get pod greeting -n pods-learn -o jsonpath='{.status.phase}{"\n"}'
```

After successful execution, the log contains `Pod work complete` and the last command prints `Succeeded`. The wait timeout bounds how long the client waits; it is not a Pod lifetime setting. This container is supposed to stop, so waiting for `Ready` would ask the wrong question. See the kubectl wait reference.

With `Always`, that same successful exit would trigger another container start. A growing restart count does not by itself prove an application crash: the restart policy and exit code explain why another instance was started.

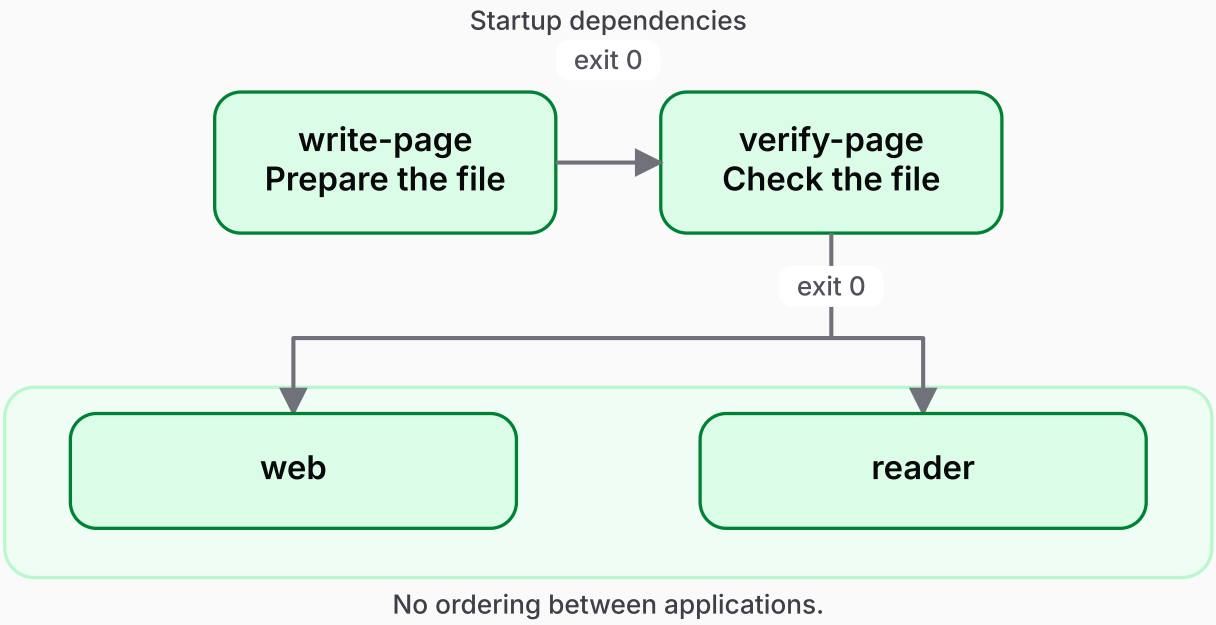
For a reusable file, the same creation command can generate YAML instead of submitting a Pod:

```bash
kubectl run greeting -n pods-learn --image=busybox:1.37.0 \
  --restart=Never --dry-run=client -o yaml \
  --command -- sh -c 'printf "Pod work complete\n"' > greeting.yaml
```

This writes a local starting manifest and leaves the existing Pod alone. Inspect the generated `spec.restartPolicy` and `containers[].command` to connect the flags to API fields. This is an alternative creation workflow; it is not needed to continue the current demonstration. The kubectl run reference documents these flags.

Init containers turn prerequisites into an ordered startup

Suppose a web process requires a prepared file. If preparation and serving start as ordinary application containers, the server can start before the file exists. A regular init container solves that ordering problem: the kubelet waits for it to exit successfully before starting the next init container, then starts the application containers.



Successful init completion releases application startup

Each regular init container must finish successfully. After initialization, both applications may start; their list order does not establish a dependency.

Init containers live under `spec.initContainers`. Each can use its own image and tools, keeping preparation tools out of the application's image. Their scripts should be **idempotent**: safe to run again. Retries or re-execution must not corrupt partially prepared work. Do not make an init container wait for an application container in the same Pod: that application is waiting for initialization to finish. See init container behavior.

Prepare a page, then run two consumers

Save the following complete manifest as `page-demo.yaml`. It uses the namespace created above; the `greeting` Pod is not a dependency.

page-demo.yaml · yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: page-demo
```

```
  namespace: pods-learn
spec:
  restartPolicy: Always
  initContainers:
    - name: write-page
      image: busybox:1.37.0
      command: ["sh", "-c", "printf 'Prepared before startup\n' > /work/index.html"]
      volumeMounts:
        - name: page
          mountPath: /work
    - name: verify-page
      image: busybox:1.37.0
      command: ["sh", "-c", "test -s /work/index.html && echo 'Page verified'"]
      volumeMounts:
        - name: page
          mountPath: /work
          readOnly: true
  containers:
    - name: web
      image: busybox:1.37.0
      command: ["httpd", "-f", "-p", "8080", "-h", "/www"]
      volumeMounts:
        - name: page
          mountPath: /www
          readOnly: true
    - name: reader
      image: busybox:1.37.0
      command: ["sh", "-c", "cat /shared/index.html; exec sleep 3600"]
      volumeMounts:
        - name: page
          mountPath: /shared
          readOnly: true
  volumes:
    - name: page
      emptyDir: {}
```

`write-page` creates the content. Its `>` redirection overwrites the demonstration file if it already exists. `verify-page` checks that it is nonempty and prints a confirmation. Only after both exit successfully may `web` and `reader` start. The HTTP server runs in the foreground because of `-f`; `reader` prints the file and then stays alive for observation. Its sleep eventually ends, at which point `Always` restarts that container.

The volume named `page` connects `/work`, `/www`, and `/shared`. Those are different paths in different containers, all mounted from the same Pod volume. `emptyDir` is temporary Pod storage: its files survive an individual container restart but are discarded

when the Pod is removed from its node. This is enough storage context for the example; persistent storage starts in post 33. The official shared-volume walkthrough describes this mounting relationship.

The second container illustrates a companion process, not another replica of the server. Listing `web` before `reader` under `containers` does not establish startup ordering between them. Both rely on init completion; neither depends on which application container starts first.

Submit the manifest, then wait for its Pod readiness condition before inspecting it:

```bash
kubectl apply -f page-demo.yaml
kubectl wait -n pods-learn pod/page-demo --for=condition=Ready --timeout=120s
kubectl get pod page-demo -n pods-learn -o wide
```

After normal startup, the table should show `2/2` ready application containers and `Running`, plus the assigned node and Pod IP. The two completed regular init containers are not counted as running application containers. A fast startup may finish before you ever see an `Init:...` display. No health probes are configured, so this readiness observation does not perform an HTTP check.

Read the startup evidence

When an administrator needs to see which stage finished, `describe` groups init containers, application containers, conditions, and recent events:

```bash
kubectl describe pod page-demo -n pods-learn
```

Under `Init Containers`, both preparation steps should have `State: Terminated`, `Reason: Completed`, and `Exit Code: 0`. Under `Containers`, `web` and `reader` should be running. In conditions, `Initialized: True` connects successful preparation to application startup. The events list may show scheduling, pulling, creating, and starting; its exact entries and timestamps depend on caching and cluster timing.

For a focused view of identity and state, read the API fields directly:

```bash
kubectl get pod page-demo -n pods-learn \
  -o custom-columns='NAME:.metadata.name,UID:.metadata.uid,PHASE:.status.phase,NODE:.spec.nodeName'
kubectl get pod page-demo -n pods-learn -o yaml
```

In the YAML, look at `status.initContainerStatuses` for preparation history and `status.containerStatuses` for application history. Each entry identifies its container by name. `state` is the current state; `lastState` can report the previous execution after a restart, and `restartCount` reports restarts observed for that container. The UID column lets you distinguish a changed container history from a different Pod object.

Choose the container whose evidence you need

Logs belong to containers. Use `-c` explicitly so the same command pattern works reliably when a Pod has several containers:

```bash
kubectl logs -n pods-learn page-demo -c verify-page
kubectl logs -n pods-learn page-demo -c reader
```

After normal startup, these print `Page verified` and `Prepared before startup`, respectively. The completed init container's logs remain useful after it exits. `write-page` redirected its output into a file, so an empty log from that container would be expected. Logs collect

standard output and standard error, not arbitrary files inside the container.

To inspect the shared network from the reader container once the web server is listening, run its bundled HTTP client:

```bash
kubectl exec -n pods-learn page-demo -c reader -- \
  wget -qO- http://127.0.0.1:8080
```

The response should contain `Prepared before startup`. The client runs in `reader`; the server process runs in `web`. They connect through the Pod's loopback interface, so this request needs neither another Pod's address nor a Service. If issued at the very instant of startup, the request can precede the server opening its socket; `Ready` without probes does not rule that out. Post 05 adds application-aware readiness, and posts 27–29 develop networking and Services.

`exec` starts an additional command inside a running container; it does not replace that container's main process. The `--` separates kubectl flags from the command being executed. A completed init container cannot accept a new `exec` command; inspect its logs and recorded exit status instead.

If an application container later restarts, an administrator can ask for the previous instance's logs:

```bash
kubectl logs -n pods-learn page-demo -c reader --previous
```

Use this only when a previous instance exists; a fresh Pod has no previous log to return. It is a view of the previous container execution, not a history of deleted replacement Pods. Wider log and event workflows belong to post 36.

A native sidecar starts early and keeps running

“Sidecar” describes a supporting role. Older manifests often express that role as another entry under `containers`, like a file reader or log forwarder. Such a companion follows ordinary application-container lifecycle rules.

A **native sidecar**, stable since Kubernetes 1.33, is an entry under `initContainers` with its own `restartPolicy: Always`. The kubelet proceeds after that container is marked started, while the sidecar continues alongside the application. A regular init container must finish successfully instead. See native sidecar lifecycle.

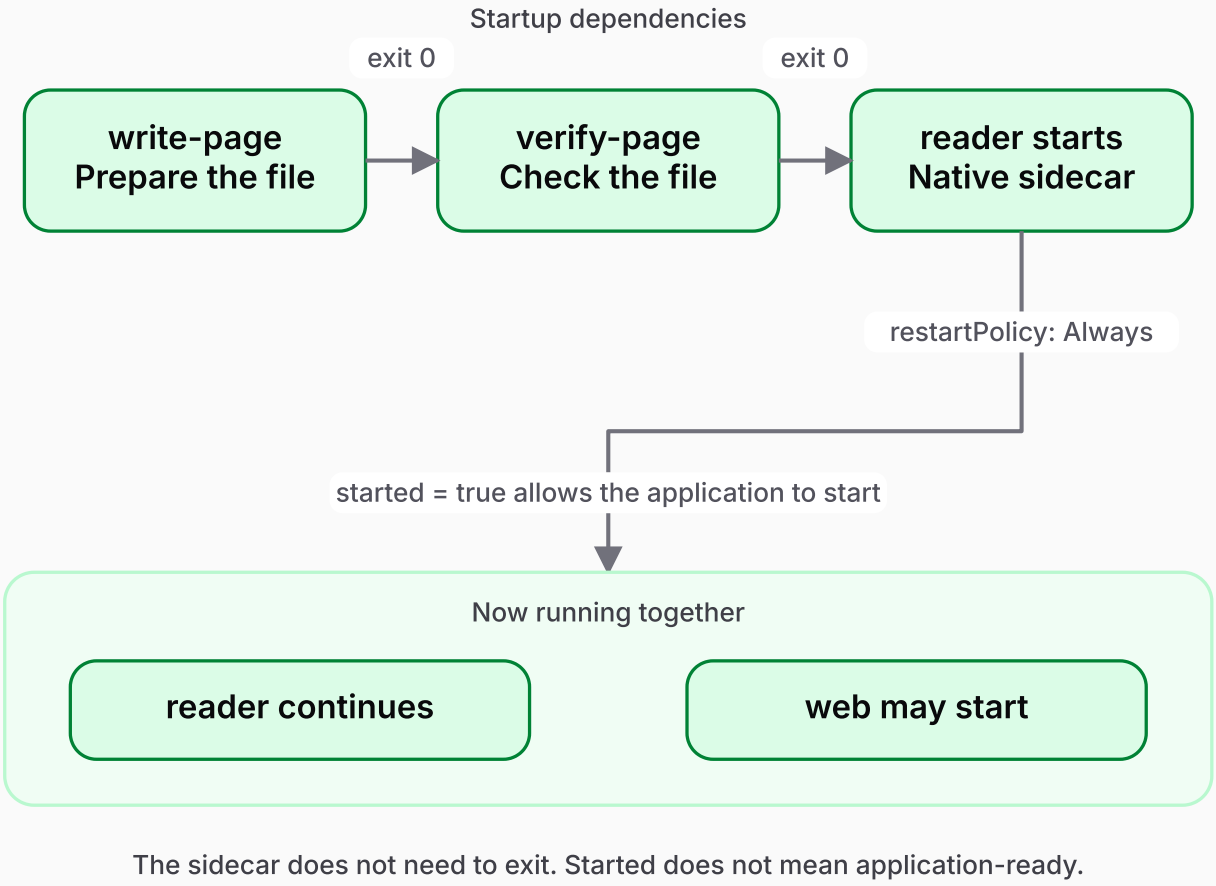
For a concrete variation, copy `page-demo.yaml` to `page-with-sidecar.yaml`, change `metadata.name` to `page-with-sidecar`, and move the entire `reader` entry out of `containers` to the end of `initContainers`, after `verify-page`. Add the container-level restart policy shown below. Keep `web`, the two preparation steps, and the volume unchanged.

reader entry under spec.initContainers · yaml

```yaml
- name: reader
  image: busybox:1.37.0
  restartPolicy: Always
  command: ["sh", "-c", "cat /shared/index.html; exec sleep 3600"]
  volumeMounts:
    - name: page
      mountPath: /shared
      readOnly: true
```

This is a replacement container entry, not a standalone manifest. The separate Pod name matters: moving containers between lists changes fields that cannot be edited in place on an existing Pod.

The resulting startup relationship is:



A native sidecar starts in sequence and continues alongside the app

The gate changes from successful completion to started = true for the native sidecar. Reader keeps running alongside web rather than blocking startup until it exits.

Create the variation and inspect the container categories:

```bash
kubectl apply -f page-with-sidecar.yaml
kubectl wait -n pods-learn pod/page-with-sidecar --for=condition=Ready --timeout=120s
kubectl describe pod page-with-sidecar -n pods-learn
kubectl logs -n pods-learn page-with-sidecar -c reader
```

`reader` now appears in `Init Containers` as running, while `write-page` and `verify-page` are terminated successfully. Its log still contains the prepared page. This difference in placement and lifecycle is what makes it a native sidecar; the container name itself has

no special meaning.

Without a startup probe, “started” does not establish application-level readiness. Probes are the next lesson. The sidecar's `Always` applies independently of the Pod-level restart policy. During graceful shutdown, the kubelet stops native sidecars after the main application containers, in reverse sidecar order. Ordinary companions do not receive that special shutdown ordering. See the sidecar documentation.

Deletion requests a shutdown, not a restart

An administrator removes these direct Pods by name after observing them:

```bash
kubectl delete pod greeting page-demo -n pods-learn
kubectl delete pod page-with-sidecar -n pods-learn --ignore-not-found
```

For a normally running Linux Pod, deletion records the termination request. The kubelet asks the runtime to stop the containers, ordinarily using `SIGTERM` so the main processes can exit cleanly. The default `terminationGracePeriodSeconds` is 30; remaining processes are forcibly killed when the grace period expires. Images can configure a different stop signal. A configured `preStop` hook runs before the stop signal and consumes the grace period; it is not extra shutdown time. See Pod termination.

`Terminating` describes this deletion process, not a sixth phase. Even `restartPolicy: Always` does not undo a deletion request or recreate the Pod. The temporary `page` volume disappears with its Pod. Because these are direct Pods, no replacement will appear automatically.

If the namespace was created solely for this walkthrough and contains nothing else you need, remove it too:

```bash
kubectl delete namespace pods-learn
```

Namespace deletion removes the resources inside it. To repeat the walkthrough, recreate the namespace and submit the manifests again; the new Pods receive new UIDs.

Misconceptions that make Pod behavior confusing

- **"Running means healthy."** Phase describes lifecycle progress. Readiness is a separate signal, and meaningful health checks require the probes in post 05.
- **"A terminated container failed."** Read its exit code. Successful init steps and finite workloads are supposed to terminate.
- **"Always means repeat every init step forever."** Successful regular init steps finish; application restarts ordinarily leave completed initialization alone.
- **"The container list is a startup script."** Ordinary application-container order provides no dependency guarantee. Use init completion for prerequisites.
- **"Same Pod means same filesystem."** Sharing files requires mounting a shared volume; sharing localhost is a different mechanism.
- **"Changing YAML changes any live Pod field."** Most Pod spec fields cannot be updated in place. For these direct demonstrations, use a new name or deliberately delete and recreate the Pod when changing its structure. Some fields, including container images, are mutable; the Pod update rules explain that distinction.

Where this fits in the CKA path

The object model from post 03 is now concrete: metadata identifies the Pod, spec defines its containers and startup rules, and status records what actually happened.

Next is post 05, Container Health: Startup, Liveness, and Readiness Probes. Post 06 adds ReplicaSets and Deployments for replacement and scaling; post 08 introduces controllers for finite workloads. Scheduling, networking, storage, and systematic troubleshooting receive their own later lessons. Here, the essential unit is still one Pod and the lifetimes of the containers inside it.

What to remember

- **One Pod identity, several container lifetimes.** A container restart keeps the Pod; replacement creates a new UID.
- Read the right layer: Pod phase for the broad lifecycle, container state for execution, conditions for specific progress and readiness signals.
- A restart policy governs exits. A successful exit can be the desired result.
- Regular init containers finish in order before applications start. Native sidecars start in that ordered list and keep running.
- Containers share a Pod network; files are shared only through explicit mounts. Use `-c` to select the container for logs and commands.
- Observe the whole chain: manifest intent, init completion, running containers, application output, then normal shutdown.