Omar Latrach

TECHNICAL ARTICLE

PUBLISHED · SEPTEMBER 9, 2026

Container Health: Startup, Liveness, and Readiness Probes

Build a clear mental model for Kubernetes startup, liveness, and readiness probes, then observe each signal through one reproducible Pod demonstration.

This is Learn post 05 of the 54-post Certified Kubernetes Administrator (CKA) preparation path. Post 04 separated a Pod's lifetime from the lifetimes of its containers. Health probes add three application-level signals to that model: has this container started, should it receive traffic, and is restarting it the right recovery action?

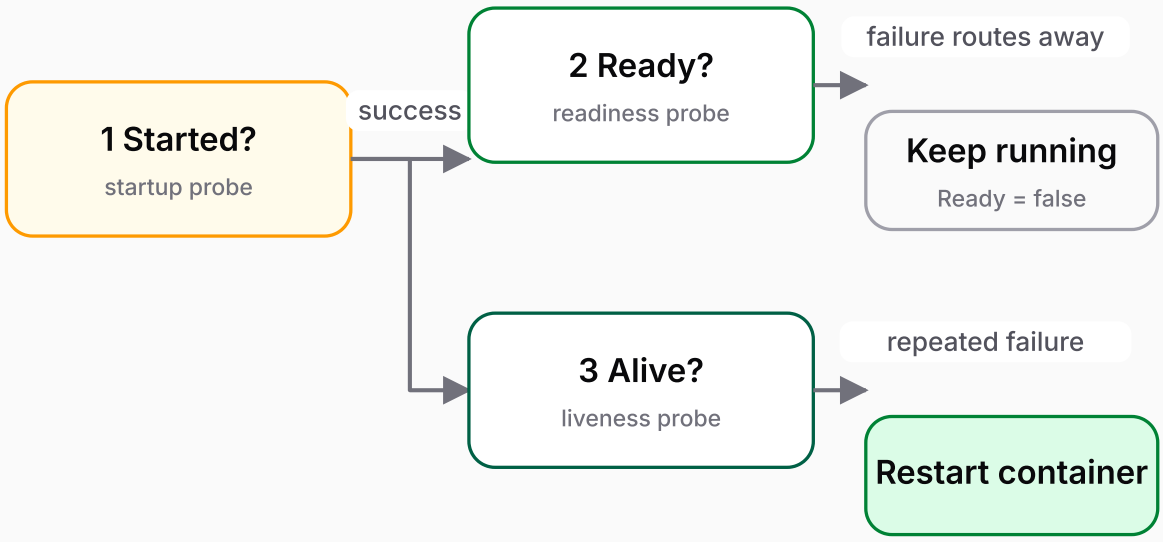
A running process is not necessarily a usable application. Because the container runtime can only report that a process exists, the **kubelet** performs probes and turns their results into Kubernetes actions.

What you'll learn

- Choose startup, readiness, and liveness probes by the action you want Kubernetes to take.
- Explain how the kubelet executes HTTP, TCP, command, and gRPC checks.
- Configure all three probes and reason about their timing thresholds.
- Observe startup gating, a readiness transition, and a liveness-triggered restart with kubectl.
- Avoid probe designs that cause false failures or restart storms.

Three questions, three different consequences

Do not memorize probes as three nearly identical YAML fields. Remember the question each probe asks and the consequence of a failed answer.



After startup succeeds, readiness and liveness run independently.

The container probe decision model

Startup is a temporary gate. After it opens, readiness controls routing eligibility while liveness controls container restarts.

Startup: has the application finished starting?

A startup probe protects a slow-starting application. While it has not succeeded, the kubelet does not run that container's readiness or liveness probes. This prevents an aggressive liveness check from killing legitimate startup work.

Startup is not a permanent health state. After one success, the startup probe stops and the gate stays open for that container instance. If startup keeps failing until its threshold is reached, the kubelet terminates the container and applies the Pod's restart policy.

Readiness: should this container receive traffic now?

A failed readiness probe marks the container not ready. The container keeps running and the kubelet keeps checking it, so a temporary condition can recover without a restart. When a Pod is not ready, matching Kubernetes Services do not use it for normal

traffic.

Readiness runs throughout the container's lifetime. It can hold traffic during initial warm-up and withdraw the Pod later during a temporary overload or dependency problem. Service and EndpointSlice mechanics belong to posts 28 and 29; here, remember only that readiness supplies the routing signal.

Liveness: is restarting this container the correct recovery?

A liveness probe detects an application that still has a process but cannot make progress, such as a deadlock. After consecutive failures reach the configured threshold, the kubelet terminates that container. Its restart policy then determines whether it starts again.

> The shortest useful rule: **startup delays judgment, readiness redirects traffic, liveness restarts the container.**

Where probes fit in the Pod model

Probes are configured per container, under an entry in spec.containers. The kubelet on the assigned node performs them. This is local container supervision: a liveness failure restarts the failed container inside the same Pod; it does not create a replacement Pod.

That distinction connects directly to post 04. The Pod keeps the same name, unique identifier (UID), node assignment, and Pod-level lifetime while the container's `restartCount` increases. Post 06 introduces controllers that can create replacement Pods; a probe alone does not do that.

Pod phase and readiness answer different questions. A Pod can be `Running` because its containers are executing while showing `0/1` in the READY column because the application is not ready for traffic. Without a readiness probe, Kubernetes has no

application-level readiness test for that container; once started, it is treated as ready unless another Pod-level condition prevents readiness.

One Pod that makes every outcome visible

The following manifest runs NGINX and exposes three training-only health files. The container waits 12 seconds before creating them, which makes the startup gate visible. Removing the readiness or liveness file later makes the corresponding HTTP request return 404.

probe-demo.yaml · yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: probe-demo
  namespace: probe-lab
  labels:
    app: probe-demo
spec:
  restartPolicy: Always
  containers:
    - name: web
      image: nginx:1.27-alpine
      command: ["/bin/sh", "-c"]
      args:
        - |
          root=/usr/share/nginx/html
          rm -f "$root/started" "$root/ready" "$root/live"
          (sleep 12; touch "$root/started" "$root/ready" "$root/live") &
          exec nginx -g 'daemon off;'
      ports:
        - name: http
          containerPort: 80
      startupProbe:
        httpGet:
          path: /started
          port: http
        periodSeconds: 2
        failureThreshold: 10
      readinessProbe:
        httpGet:
          path: /ready
          port: http
        periodSeconds: 2
        failureThreshold: 1
        successThreshold: 1
      livenessProbe:
        httpGet:
```

```
    path: /live
    port: http
  periodSeconds: 2
  timeoutSeconds: 1
  failureThreshold: 3
```

These files are controllable switches for learning, not a production health design. A real endpoint should test the application property named by the probe without performing expensive work or changing state.

Create the namespace and Pod

Create an isolated namespace, then apply the declarative manifest. The namespace command is imperative because the namespace itself is only supporting this demonstration.

bash

```
kubectl create namespace probe-lab
kubectl apply -f probe-demo.yaml
kubectl get pod -n probe-lab probe-demo --watch
```

After scheduling and image startup, the important transition looks like this. Pull time and ages vary by cluster.

plaintext

```
NAME         READY   STATUS    RESTARTS   AGE
probe-demo   0/1     Running   0          4s
probe-demo   1/1     Running   0          14s
```

The first line is the distinction to notice: the process is running, so the Pod phase is Running, but the startup gate has not opened and the container is not ready. After /started returns HTTP 200, startup succeeds; readiness and liveness begin; /ready succeeds; and READY becomes 1/1.

Read the underlying status, not only the table

A custom-columns view connects the concise READY value to container status fields. Run it after the Pod becomes ready.

```bash
kubectl get pod -n probe-lab probe-demo \
  -o custom-columns='NAME:.metadata.name,PHASE:.status.phase,STARTED:.status.containerStatuses[0].started,READY:.status.containerStatuses[0].ready,RESTARTS:.status.containerStatuses[0].restartCount'
```

```plaintext
NAME         PHASE     STARTED   READY   RESTARTS
probe-demo   Running   true      true    0
```

STARTED records startup-probe success for this container instance. READY records its current readiness. RESTARTS counts container restarts within this Pod. In a multi-container Pod, every regular container must be ready before the Pod's Ready condition becomes true.

Use describe when you need probe configuration, container state, conditions, and recent events together. Early startup failures are expected in this demonstration because /started intentionally returns 404 before the 12-second delay finishes.

```bash
kubectl describe pod -n probe-lab probe-demo
```

A readiness failure changes routing state, not process state

Remove only the readiness file, then watch the Pod. The next readiness check receives HTTP 404.

```bash
kubectl exec -n probe-lab probe-demo -- \
  rm /usr/share/nginx/html/ready

kubectl get pod -n probe-lab probe-demo --watch
```

```plaintext
NAME         READY   STATUS    RESTARTS   AGE
probe-demo   0/1     Running   0          1m
```

The Pod remains Running and RESTARTS remains zero. Because the failure threshold is one, one failed readiness check is enough to set the container's ready field false. The kubelet continues probing instead of killing the process.

Restore the file. A later successful readiness check returns the same container to Ready without a restart.

```bash
kubectl exec -n probe-lab probe-demo -- \
  touch /usr/share/nginx/html/ready

kubectl get pod -n probe-lab probe-demo
```

A liveness failure restarts the container, not the Pod

First record the Pod UID. Then remove /live so the liveness probe repeatedly receives HTTP 404.

```bash
kubectl get pod -n probe-lab probe-demo \
  -o jsonpath='{.metadata.uid}{"\n"}'

kubectl exec -n probe-lab probe-demo -- \
  rm /usr/share/nginx/html/live

kubectl get pod -n probe-lab probe-demo --watch
```

With a two-second period and failure threshold of three, the kubelet acts after three consecutive failed checks. Probe scheduling and graceful container termination mean the observed wall-clock time is not an exact six-second timer.

```plaintext
NAME        READY   STATUS    RESTARTS   AGE
probe-demo   0/1     Running   1          2m
probe-demo   1/1     Running   1          2m
```

RESTARTS increases but the Pod UID does not change. The new container instance runs the command again, waits 12 seconds, and recreates all three files. During that wait, its startup probe gates readiness and liveness again; that explains the temporary 0/1 before recovery.

Confirm the original UID and inspect the event trail. Event ages and wording vary slightly by Kubernetes version, but the useful reasons are Unhealthy followed by Killing.

```bash
kubectl get pod -n probe-lab probe-demo \
  -o jsonpath='{.metadata.uid}{"\n"}'

kubectl describe pod -n probe-lab probe-demo
```

```plaintext
Events:
  Type     Reason     Message
  ----     ------     -------
  Warning  Unhealthy  Liveness probe failed: HTTP probe failed with statuscode: 404
  Normal   Killing    Container web failed liveness probe, will be restarted
```

For a real failure, pair events with application evidence. After a restart, `kubectl logs --previous` requests the terminated container instance's logs instead of only the new instance's logs.

```bash
kubectl logs -n probe-lab probe-demo -c web --previous
```

Choose the check mechanism by what it proves

- HTTP GET checks an application endpoint on the Pod IP. Status codes from 200 through 399 succeed. It is usually the clearest choice for HTTP applications because the endpoint can express application-level health.
- TCP socket checks whether the kubelet can open the container's port at the Pod IP. An open port proves less than a correct application response, but it fits servers without an HTTP health endpoint.
- Exec runs a command inside the container. Exit code 0 succeeds; any non-zero exit code fails. Use a small, dependable command and remember that repeated process creation has overhead.
- gRPC invokes the standard gRPC Health Checking Protocol. It succeeds when the service reports SERVING and fits applications that implement that protocol directly.

Each probe defines exactly one mechanism. HTTP and TCP probes can use a named container port, as the manifest uses http. A gRPC probe requires a numeric port. Probes target the container or Pod directly, not a Service name.

Tune a failure window, not a single magic delay

The same timing fields appear on all three probe types, but their safe values depend on application behavior and on the consequence of failure.

- initialDelaySeconds waits before the first probe. When a startup probe exists, readiness and liveness are already gated until startup succeeds, and their initial delays begin after that success.
- periodSeconds controls how often the kubelet checks. The default is 10 seconds.
- timeoutSeconds limits one check. Its default is only one second, which is easy to leave accidentally too strict.
- failureThreshold is the number of consecutive failures required for the probe to fail overall. A later success resets the consecutive-failure count.
- successThreshold is the number of consecutive successes needed after failure. It must be one for startup and liveness probes; readiness may use a higher value to avoid rapid flapping back to ready.

A useful first estimate for startup tolerance is failureThreshold × periodSeconds. The demonstration allows about 20 seconds of checks for a 12-second startup. Treat that multiplication as a configuration budget, not an exact stopwatch; initial delay, individual timeouts, scheduling, and termination add context.

Common probe mistakes

- Using liveness for a temporary dependency failure. If a database or remote API is down, restarting every client container usually cannot repair it. Report not ready when the application should stop receiving traffic; reserve liveness failure for states a restart can plausibly fix.

- Making liveness too sensitive. Under load, a slow health response can trigger restarts, reduce capacity, and increase load on the remaining Pods. Require evidence of an unrecoverable state, not one brief delay.
- Using only a long liveness initial delay for unpredictable startup. That forces one setting to serve two jobs. A startup probe gives generous startup tolerance while preserving faster liveness detection after startup.
- Assuming containerPort creates a check. It documents and names a port; it does not probe anything. Health behavior exists only when startupProbe, readinessProbe, or livenessProbe is configured.
- Treating Running as Ready. Always check the READY column, Pod conditions, container status, and events before concluding that an application is available.
- Forgetting that a bad check is still authoritative. The kubelet cannot tell whether a 404 means a sick application or a misspelled path. Verify the path, port, protocol, command, timeout, and application behavior together.

Administrator workflow

When health behavior looks wrong, move from the visible symptom toward the cause: read readiness and restart count, inspect status and events, verify the rendered probe spec, then correlate with current and previous application logs.

```bash
kubectl get pod -n <namespace> <pod> -o wide
kubectl describe pod -n <namespace> <pod>
kubectl get pod -n <namespace> <pod> -o yaml
kubectl logs -n <namespace> <pod> -c <container>
kubectl logs -n <namespace> <pod> -c <container> --previous
```

This is an observation sequence, not a guarantee that every command will be useful in every incident. The broad troubleshooting framework returns in post 36, and the later Practice phase applies it to broken resources.

Clean up and continue

The namespace contains only this demonstration, so deleting it removes the Pod and all related namespaced state.

```bash
kubectl delete namespace probe-lab
```

For exact current semantics, keep the official probe concepts and probe configuration guide close while administering a cluster.

What to remember

- The kubelet performs probes per container; probes are not application monitoring performed by the control plane.
- Startup protects initialization and gates readiness and liveness until it succeeds.
- Readiness failure keeps the container running but makes the Pod ineligible for normal Service traffic.
- Repeated liveness failure terminates the container; restart policy controls what happens next.
- A container restart increases restartCount but keeps the Pod UID. A controller-created replacement is a different Pod, which post 06 teaches next.
- Choose a mechanism that proves the right property, then tune thresholds around real startup and failure behavior.