Engineer

Health check

an automated test of whether a service is working

What it is

A health check is a small test, repeated on a schedule, that asks a running service whether it can do its job, together with a rule for what happens when the answer is no. The test might be an HTTP request to a dedicated path, a command run inside a container, or an attempt to open a TCP port. The rule is what turns it from a measurement into a control: a failed check can take a server out of rotation, restart a container, stop a deployment or wake someone up.

Docker builds the idea into the image. The `HEALTHCHECK` instruction in a Dockerfile names a command that Docker runs inside the container at a fixed interval. Exit status 0 means healthy, 1 means unhealthy, and 2 is reserved and must not be used. A container with a check begins in the state `starting`, becomes `healthy` as soon as one check passes, and turns `unhealthy` after a set number of consecutive failures. The defaults are a check every 30 seconds, a 30-second timeout and three retries; a run that exceeds its timeout counts as a failure, and its process is killed. An optional start period gives a slow application time to boot, and failures inside it are not counted. A Dockerfile holds one such instruction, the last one wins if there are several, and `HEALTHCHECK NONE` switches off a check inherited from the base image.

Kubernetes splits the question into three probes, each with its own consequence. A startup probe asks whether the application has finished starting; until it succeeds, the other two are held back, and if it keeps failing the container is killed and handled by its restart policy. A liveness probe asks whether the container should be restarted, and is meant for states such as a deadlock, where the process runs but makes no progress. A readiness probe asks whether the container should receive traffic. When it fails, nothing is restarted: the Pod’s address is taken out of the endpoints of every Service that matches it, and requests go to the other replicas until the probe passes again.

Each probe uses one of four mechanisms. An `httpGet` probe passes on any status code from 200 up to, but not including, 400. A `tcpSocket` probe passes if the port accepts a connection, even one that is closed straight away. A `grpc` probe passes when the service's gRPC health check answers `SERVING`. An `exec` probe runs a command in the container and passes on exit status 0. Unless told otherwise, Kubernetes probes every 10 seconds, waits one second for an answer, and acts after three failures in a row.

Load balancers add a second kind. The checks above are active: something sends a test request whether or not users are around. A passive check watches real traffic instead. Open-source nginx works this way: when a request to an upstream server ends in an error, nginx marks that server as failed and avoids it for a while, and with the default `max_fails` of 1 a single failed attempt is enough. The nginx documentation lists application health checks among the features of the paid NGINX Plus subscriptions. A passive check costs nothing extra, but it learns about a broken server from a user's failed request, and it says nothing about a server that receives no traffic.

Use the concept in a decision

Put "What is a health check?" to work in a real decision. We can help relate the concept to the system you are building or buying. Monitoring and alerts that speak up while a fault is still small, and the figures that show how often it stays that way.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Built the company's first observability dashboard — the company's first view of its own systems

Built the company's first observability dashboard Get in touch →

Why it matters

A running process is not a working service. Docker's reference gives the typical case: a web server caught in an endless loop, no longer taking new connections, although its process is still running. Anything that only watches whether the process exists, such as a systemd unit with a restart policy, sees nothing wrong. Only a question asked through the same door the users use can tell the difference, and that is the whole case for a health check: it replaces "is it running?" with "does it answer?".

Two decisions make a check useful or dangerous. The first is what the check tests, which amounts to a definition of "working" written in code. A handler that returns 200 without touching anything proves the web framework is up; a

handler that also queries the database proves the database is reachable as well, and fails whenever it is not. The second is what happens on failure, which sets the blast radius. Taking one replica out of rotation is cheap and reversible. Restarting it throws away its in-flight work and its warm caches, and if every replica fails the same check at the same moment, restarting all of them turns a slow dependency into a full outage.

Kubernetes warns about exactly this. Its documentation says liveness probes must be configured so that they truly indicate an unrecoverable failure, and that a wrong implementation can cause cascading failures: containers restarted under high load, failed client requests, and more work landing on the pods that remain. The advice that follows is to keep the two questions apart. When an application depends strictly on back-end services, its liveness probe should pass as long as the application itself is healthy, while its readiness probe also checks each required back end. Traffic then avoids a Pod that could only answer with errors, and the Pod is not killed for a fault that lies somewhere else.

Health checks also serve monitoring. An endpoint that a load balancer polls can be polled from outside as well, and the alert that fires when it fails is often the first sign of an incident. When Irakli Nadareishvili wrote the Internet-Draft Health Check Response Format for HTTP APIs, there was no standard for what such an endpoint returns, and most applications relied on the HTTP status code alone or on a format of their own. The draft proposed a JSON body with the media type `application/health+json` and a mandatory `status` field of `pass`, `fail` or `warn`, where `pass` and `warn` go with a 2xx or 3xx status code and `fail` with a 4xx or 5xx. Its last revision, from October 2021, expired in April 2022, and the IETF lists it as not endorsed and without formal standing. It is useful as a shared vocabulary, not as a standard.

What it looks like in practice

In a Dockerfile, the check ships with the image it checks. Here the application's own small probe program requests a path on localhost and exits with 0 or 1:

```
HEALTHCHECK --interval=15s --timeout=3s --start-period=60s --retries=3 \
  CMD ["/app/bin/probe", "http://127.0.0.1:8080/livez"]
```

Docker runs the probe every 15 seconds and gives each run three seconds. Failures during the first minute do not count unless a check has already passed, and on Docker Engine 25.0 or later the checks in that minute run every five seconds, the default start interval, so a quick start is noticed quickly. After three failures in a row the container is marked `unhealthy`. The first 4096 bytes of the probe's output are kept for `docker inspect`, and every change of status emits a `health_status` event. The reference describes a status, not an action: something else has to act on it.

In Kubernetes, the three probes can be given different paths, so that each asks its own question:

```
containers:
  - name: api
    ports:
      - containerPort: 8080
    startupProbe:
      httpGet: { path: /livez, port: 8080 }
      periodSeconds: 5
      failureThreshold: 24 # up to 2 minutes to start
    livenessProbe:
      httpGet: { path: /livez, port: 8080 }
      periodSeconds: 10
      timeoutSeconds: 2
      failureThreshold: 6 # about a minute before a restart
    readinessProbe:
      httpGet: { path: /readyz, port: 8080 }
      periodSeconds: 5
      timeoutSeconds: 2
      failureThreshold: 2
```

`/livez` answers for the process alone, and `/readyz` also checks the database and anything else a request cannot do without. The startup probe uses the liveness path, as the Kubernetes documentation recommends, and allows 24 × 5 seconds before the container is given up on. Once it passes, liveness takes over with a long fuse, and readiness with a short one: about ten seconds of failures take the Pod out of the Service, and a minute of them restarts it. The defaults would allow three failures ten seconds apart, which is too little for an application that needs two minutes to load its data. The documentation's own rule is that a container which usually takes longer than the initial delay plus failureThreshold × periodSeconds to start needs a startup probe.

A health check can also be turned inside out. In dead-man's-switch host monitoring, from September 2026, a 464 MB host running a forge and a web server checks itself every fifteen minutes, with jitter: a small script measures available memory and free disk and pings an external service only when both are above their thresholds. Silence is the alert. A host that has run out of memory, lost its network or stopped booting sends the same signal as one that is merely unhealthy, which is why this shape was chosen over an agent that reports a status. The check reads available memory rather than free memory, because the kernel fills idle memory with cache, and six variable names that said "free" were corrected so that nobody would "fix" a working alarm into one that breaches permanently. A received signal confirms that the script ran and its available-memory and free-disk thresholds passed; it does not certify general host health or that the site is serving. Monitoring can expose warning signs, although the timing depends on the workload and the conditions observed.

A check at startup answers yet another question. In email with failover and delivery logging, from 2025, mail runs through three providers with automatic failover, and in front of it a startup check sends a real message through the whole stack, behind a flag. In development a failure logs a warning and the application carries on; in staging and production it is fatal, and the process exits rather than deploy a build that cannot send mail. The page sums up the result as a whole category of silent failure moved from "a user notices days later" to "the deploy stops".

Where it goes wrong

The check proves only that the process answers. A handler that returns 200 without touching anything stays green while the database is unreachable, the disk is full or every real request ends in an error. Let the readiness check exercise what a request needs, cheaply, and add a check from outside that follows a user's path through DNS, TLS and the load balancer, since the kubelet sends its probes straight to the Pod's own IP address and sees none of that.

Liveness depends on something outside the container. If the liveness probe queries the database, a slow database fails it in every replica at once, and Kubernetes restarts them all at the moment the service is already struggling. Let liveness answer for the process alone, as the Kubernetes documentation advises, and put dependencies in readiness, where a failure diverts traffic instead of killing containers.

The timings do not fit the application. Kubernetes waits one second for an answer by default, so an endpoint that slows down under load fails while the application is merely busy, and a container that starts more slowly than its liveness probe allows is killed before it is ready, again and again. Measure how long the endpoint takes under load and set the timeout above it, and give slow starters a startup probe sized to the slowest real start.

A redirect passes. An HTTP probe in Kubernetes counts any status from 200 to 399 as success, so a path that redirects to a login page looks healthy. If the redirect points to another host, the kubelet does not follow it and records the probe as successful, with a warning. With the HTTPS scheme the kubelet also skips certificate verification, so an expired certificate never fails a probe. Point probes at a path that answers 200 directly and returns a 5xx when something is wrong, and check certificates from outside.

The health endpoint says too much to too many. A detailed response that lists every dependency, version and host name informs an attacker as well as a load balancer, and the Internet-Draft's security considerations note that health endpoints may need authentication and role-based access control. Keep the public answer to a status, and serve the details on an internal port or to authenticated callers only.

The check costs more than it should. Kubernetes warns that `exec` probes fork processes on every run, which on densely packed nodes with short intervals can add CPU overhead to the node, and a probe that runs a heavy query every few seconds on every replica is load in its own right. Prefer HTTP, TCP or gRPC probes, keep the endpoint cheap, and cache the result of an expensive dependency check for a few seconds instead of repeating it for every probe.

Further reading

- Dockerfile reference: HEALTHCHECK — the instruction, its options and defaults, and what each exit status means.
- Liveness, Readiness, and Startup Probes — what each probe type does, the four check mechanisms and every configuration field with its default.
- Configure Liveness, Readiness and Startup Probes — worked examples, including a startup probe that gives a slow application five minutes.
- Health Check Response Format for HTTP APIs — the expired Internet-Draft behind `application/health+json` and its pass, fail and warn statuses.

- Using nginx as HTTP load balancer — the passive, in-band health checks in open-source nginx and the `max_fails` and `fail_timeout` settings.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is a health check?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/health-check/