Engineer

Daemon

a program that runs in the background, unattended

What it is

A daemon is a long-lived process that no person starts by hand: the operating system or a service manager launches it, it holds no terminal, and it stays alive to answer requests or keep watch over some part of the machine. The plural is daemons, and many of them carry the word in their name as a trailing letter d: `sshd` accepts SSH connections, `crond` runs scheduled jobs, `systemd-journald` gathers logs. Windows calls the same kind of program a service, and so does systemd, which supervises most daemons on Linux today.

The traditional Unix recipe, which the Linux `daemon(7)` manual still sets out in fifteen numbered steps, has the program detach itself. It closes inherited file descriptors, resets signal handlers, forks, calls `setsid()` to leave the terminal's session, and forks again so that it can never reacquire a terminal. The first child exits so that PID 1 adopts the survivor, which points its standard streams at `/dev/null`, moves to `/`, writes a PID file and drops privileges. The C library's `daemon()` call does part of this, and the manual advises against it because it implements only a subset of the steps.

The new-style daemon described in the same manual, the kind systemd and Apple's launchd both expect, turns the recipe inside out. The program stays in the foreground, because the supervisor already started it detached, with a clean environment, the right user and the right working directory. It writes its log lines to standard error, where the service manager collects them, says when it is ready to serve, exits cleanly on SIGTERM and reloads its configuration on SIGHUP. Apple's guide to launchd jobs is blunt about the old habit: a job must not daemonize itself, and if it forks and exits, launchd concludes that the process has died.

Two neighbours get confused with it. A scheduled job, run by cron or a systemd timer, starts, does its work and exits, whereas a daemon stays. And macOS splits the word in two: a launch daemon serves the whole system and is loaded at boot, while a launch agent belongs to one user and runs only while that user is logged in. Linux draws the same line with systemd's per-user instances.

Use the concept in a decision

Put "What is a daemon?" to work in a real decision. We can help relate the concept to the system you are building or buying. Day-to-day care of servers, mail, accounts and updates, so the things everybody relies on stay boring.

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. Ran 10 PostgreSQL and MS SQL servers on Ubuntu — 10 database servers deployed and kept healthy

Ran 10 PostgreSQL and MS SQL servers on Ubuntu Get in touch →

Why it matters

Each daemon on a server is a standing commitment. It holds memory whether or not anyone uses it, and somebody has to patch it whenever its vendor ships a fix. And if it listens on a network port, it is part of the attack surface for every hour it runs. So the useful question is rarely which daemon to install; it is whether the work needs a resident process at all.

Three shapes cover most work. Something that happens on a schedule, such as a backup, a report or a certificate renewal, needs no resident process: a timer starts it, it runs to completion, and between runs it costs nothing. Something that answers occasional requests can be socket-activated: the service manager owns the listening socket and starts the daemon on the first connection; the socket stays bound across restarts, so incoming requests queue instead of being refused. Only work that has to react continuously, a web server, a database, a mail transfer agent, earns a process that is always up. On a small machine that choice decides whether everything fits in memory.

A second decision comes with every third-party daemon headed for production, and four questions settle it. Will it stay in the foreground when asked to? Does it log to standard error, or insist on log files of its own? Can it tell the supervisor that it is ready, rather than merely started? And does it need root, or can the service manager hand it the one privileged thing it needs, such as a socket on

port 443? One that answers all four well is cheap to run for years; one that fails them arrives with a wrapper script, its own log rotation and an exception in the next security review.

What it looks like in practice

Serving the site over Gemini and Gopher cost two resident processes, not a second build. When the site was mirrored to both protocols, the build that writes the HTML also wrote 777 Gopher menus and the same number of Gemini documents at the recorded implementation stage, and two small daemons on the same host took on serving them. A separate build comparison recorded zero HTML bytes changed when the output formats were added; that result does not describe later editorial or visual changes.

The Tor onion mirror shows the network side: its daemon builds outbound circuits, so nothing dials in and no firewall port opens. The role that sets up the onion mirror installs the daemon from the distribution's own package, writes a configuration with the outbound proxy port switched off, and declines to overwrite an identity it did not generate. One step taught a general lesson: the key landed correctly, the converge reported no change, and the new address went unpublished, because at that point nothing in the configuration named the new directory. The role now fails whenever an identity sits on disk with no service naming it.

The reporting shows what a quiet daemon can hide. Six read-only collectors that render the host as Markdown were reworked once it was clear that a blind collection and a healthy host printed the same page: a mirror whose daemon changed its log format reported the same zero as a mirror nobody visited. The collectors now report, among other things, journal lines read against journal lines recognised, so a daemon that changes what it writes shows up as a gap rather than as calm.

Where it goes wrong

The service starts and stops again a second later. An old-style daemon forks and lets its parent exit, as the recipe says. Under systemd's default `Type=simple`, the process systemd launched is the main process, so its exit means the service has stopped, and with the default `KillMode=control-group` the child that was meant to carry on is killed along with everything else in the unit's control group.

Run the program with its foreground flag; for the few that cannot be changed, declare `Type=forking` with `PIDFile=`, a type the manual discourages for new code.

Clients connect too early and are refused. With `Type=simple`, systemd treats the service as started the moment it has forked the process, so units ordered after it start while the daemon is still reading its configuration. The service manual adds that `systemctl start` reports success under that type even when the binary is missing or the configured user does not exist. Prefer `Type=exec`, which waits until the binary has really been executed, and for anything others depend on, let the daemon announce readiness itself under `Type=notify`:

```
#include <systemd/sd-daemon.h>

int main(void) {
    load_config();
    open_listeners();
    sd_notify(0, "READY=1"); /* units ordered after this one start now */
    return serve_forever();
}
```

A stop script kills a stranger. A forking daemon writes its process ID to a file so that scripts can signal it later. After a crash the file stays behind, the kernel hands the same number to an unrelated process, and the next stop sends SIGTERM to the wrong program. The service manual’s advice is to avoid PID files in modern projects altogether. A supervisor that started the process tracks it through its control group and has no file to go stale.

A daemon dies at startup and leaves no trace. The old recipe points standard output and error at `/dev/null`, so an error raised before the daemon opens its own log file disappears. Under a supervisor, leave both streams alone: whatever the daemon writes to standard error lands in the journal, where `journalctl -u` reads it unit by unit.

One exploited bug hands over the whole machine. A daemon that runs as root because it needed port 80 at startup keeps root for every request it serves afterwards. Let the service manager bind the privileged socket and pass it in, and leave privilege dropping to the supervisor, as `daemon(7)` recommends; settings such as `NoNewPrivileges=` and `ProtectSystem=` narrow what a compromised process can touch.

Further reading

- daemon(7) — the fifteen steps of the classic recipe, and what a new-style daemon does instead.
- daemon(3) — the C library shortcut, and the session-leader flaw its own manual records.
- sd_notify(3) — how a daemon tells systemd that it is ready, reloading or stopping.
- systemd.socket(5) — socket activation, where the manager holds the port and starts the daemon on demand.
- setsid — the POSIX definition of the call that detaches a process from its terminal.
- Creating Launch Daemons and Agents — Apple's rules for launchd jobs, including why one must never fork and exit.

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 daemon?" 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/daemon/