Supervising Ordinary Processes: A Docker-Free Service Cluster Manager in Rust

Daozhendong Yang
School of Intelligent Science and Engineering,
Yunnan Technology and Business University, Kunming, China
hi@yxpil.com

Abstract

Maylin is a small service cluster manager, written in Rust, that supervises ordinary host processes instead of containers. A service is a command line assembled from a plugin template—a runtime executable, a script, and three placeholders—so Node, Python, and native programs share one lifecycle: six observable states, restart policies, health probes, an event log, and an HTTP/WebSocket control plane with token-based roles. This paper describes that supervision model and measures it against a real build on Windows. With `max_retries = 3`, a crashing child is launched four times, with measured gaps of 2.05, 4.07, and 6.05 s, and is then marked failed; a child that exits 0 is launched once and left exited. Stopping a child that ignores stdin takes 1.21 s with a 1 s grace period. Restarting an instance with four requests in flight loses all four when it is the only upstream, but loses none with two, at the cost of 2 duplicate side effects in 6 executions. An authorization matrix over 18 method–path pairs and 4 principals shows that every endpoint rejects anonymous callers, but also that role checks are per-route heuristics rather than per-object policy, and that several endpoints return 500 or hang instead of failing cleanly. Maylin provides no isolation, no reproducible images, and no consensus: it is a deliberate midpoint between systemd, supervisord, and Compose, and the paper is explicit about what that midpoint gives up.

Artifact. Source, tests, measurement scripts, and the JSON file from which every number in this paper is drawn are available in the maylin repository at commit `3c285fe` (2026-10-04). The scripts re-run the experiments against a freshly built binary; no number below was typed by hand.

1 Introduction

Every small deployment eventually acquires a shell script that restarts things. It starts as `while true; do ./svc; sleep 1; done`, acquires a health check, then a second host, then a password, and within a year it is an unmaintainable supervisor. The two conventional exits from this path are heavyweight in opposite directions. systemd (Poettering, 2024) supervises processes excellently but offers no HTTP control plane, no per-service roles, and no view across hosts. Docker Compose (Docker, Inc., 2024) offers reproducible images and a declarative topology, but requires a container runtime, an image build for every code change, and a networking model that the services must be taught to live inside.

Maylin[1] takes a third position: services are ordinary subprocesses of one manager daemon, described by small TOML files, controlled over HTTP, and grouped into a cluster by a static peer list. There is no container, no image, no namespace. A service sees the host's filesystem, network, and environment because that is exactly what its operator wanted. The manager adds what the shell script lacks: a lifecycle with observable states, a restart policy with backoff, an audit trail, role-based access, and a load balancer in front of replicas. The whole implementation is 4,871 lines of Rust (4,016 in `src/`, 723 in integration tests, 132 in two example programs), and builds to a single binary.

This paper documents the *supervision model* rather than the feature list, because that is where the design decisions are. Concretely:

C1. A plugin template turns a runtime into a supervised service. Eight fields describe an executable; substitution of `{script}`, `{port}`, and `{name}` produces an argument vector that is never passed through a shell, and the same lifecycle applies to Node, Python, and raw binaries (§3.2).

C2. The supervisor distinguishes crash from clean exit by one comparison—whether the exit description equals `exit code 0`—and maps that bit onto three restart policies. We show the consequences, including an off-by-one that yields $k + 1$ launches for `max_retries` $= k$ (§3.3).

C3. Cron and the supervisor share entry points but not state. A scheduled restart calls the same function as the HTTP handler and therefore inherits its lack of draining; we measure what happens to requests in flight (§3.4, §5.4).

C4. Authorization is enforced by a path-and-method heuristic at one middleware. We enumerate all 18 method–path pairs and four principals, and report both the enforcement and its gaps (§3.5, §6.1).

[1]The name is the project's; this paper is written by its author and evaluates the implementation as it stands, including the places where it falls short of its own documentation.

C5. Clustering is aggregation, not consensus. Peers are static, every node is authoritative only for its own configuration, and a peer action that cannot be parsed is silently skipped (§3.6, §6.4).

Non-goals. Maylin does not isolate services. It does not build or distribute artifacts; the code a service runs is whatever is on the host's disk at start time. It does not provide reproducible environments, and it does not reconcile desired state across nodes. It does not implement membership, failure detection, leader election, or distributed scheduling. It is not a replacement for Kubernetes and does not attempt to be one; where it overlaps with systemd, supervisord, and Compose, §8 says precisely how.

2 Background

2.1 Process supervision is a solved problem

A supervisor owns a child process and answers three questions: is it alive, should it be, and what happens when it is not. systemd answers them with units, cgroups, and a restart policy; supervisord (Supervisor Development Team, 2024) with an INI file and a per-program state machine; runit and s6 with a directory of `run` scripts; Erlang/OTP with supervision trees and restart intensities (Armstrong, 2003). The vocabulary is mature: permanent, transient, and temporary children; intensity limits; graceful stop signals; backoff. Maylin does not invent new vocabulary. Its `RestartPolicy` has exactly three values—`no`, `on-failure`, `always`—and its states are the usual ones.

What none of these tools provides is a multi-tenant control plane. systemd's D-Bus interface can be exposed over the network, but it was designed for local privilege separation, not for a team where one person may restart a service and another may only read its logs. supervisord's web interface has historically had no authentication at all. The cluster is also absent: systemd units live on one machine, and grouping them is left to configuration management. Maylin's contribution in this space is not supervision itself but the coupling of supervision to an authenticated, cluster-aware API, with the smallest possible runtime footprint.

2.2 Container orchestration

Containers package a process together with its filesystem, and isolate it using namespaces (Biederman, 2006) and cgroups (Menage, 2007), following the OCI runtime specification (Open Container Initiative, 2024). Docker (Merkel, 2014) made that packaging usable; Compose (Docker, Inc., 2024) made multi-service topologies declarative; Borg (Burns et al., 2016) and Kubernetes (The Kubernetes Authors, 2024) made scheduling across machines routine. The benefits are real and well documented: an image is a reproducible artifact, resource limits are enforced by the kernel, and a failed container can be replaced rather than repaired.

The costs are equally real. A container runtime must be installed and kept running (or, with a daemonless runtime, still invoked). Every code change requires a build and a push/pull cycle unless the source is bind-mounted, which reintroduces host dependencies. The service must be taught to log to stdout, to handle ephemeral filesystems, and to be configured through environment variables—the twelve-factor discipline (Wiggins, 2011)—or it will fight the platform. On hosts where the service needs a GPU driver, a hardware dongle, a licensed compiler, or a Windows-only agent, the container boundary is either porous or prohibitive.

2.3 The gap maylin occupies

The target deployment is three to twenty hosts, a handful of services per host, written by a small team, running on machines the team already owns. The services are conventional: a Python worker, a Node front end, a compiled daemon. The team wants a restart policy, a log tail, a button that stops one service, a password that distinguishes an operator from a reader, and one page that shows whether all replicas are up. It does not want a registry, a CNI plugin, or a control plane with its own upgrade cadence.

That is a narrow band, and it is not empty: supervisord plus a reverse proxy plus a spreadsheet is a common approximation, and for a single host it is adequate. Maylin differs by making the cluster and the access model part of the same binary, and by treating the load balancer as part of the supervision story (a restart is precisely when the balancer must route elsewhere). Whether that is worth a new implementation is a judgement the reader can make from the measurements in §5; the paper does not argue that it is universally so.

3 Design

3.1 Shape of a deployment

One `maylin` process runs per host (Figure 1). It reads `maylin.toml` for global settings (HTTP port, token table, peer list, load-balancer listeners), one file per plugin, one file per instance, and one file per schedule. Each instance becomes a long-lived asynchronous task that owns the child process; the task, not the process table, is the unit of supervision. An HTTP server exposes the control plane; a WebSocket endpoint attaches to an instance's output and input; a load-balancer listener forwards TCP requests to a set of upstream instances, which may be local or on peers.

The configuration is the source of truth, and it is read at start and on explicit reload. There is no database and no distributed store; the daemon writes an event log (JSON lines) and a small status snapshot, both of which are outputs rather than inputs. This choice is what allows a host to be understood by reading five small TOML files, and it is also why there is no reconciliation: a config file edited on disk does nothing until someone calls reload.

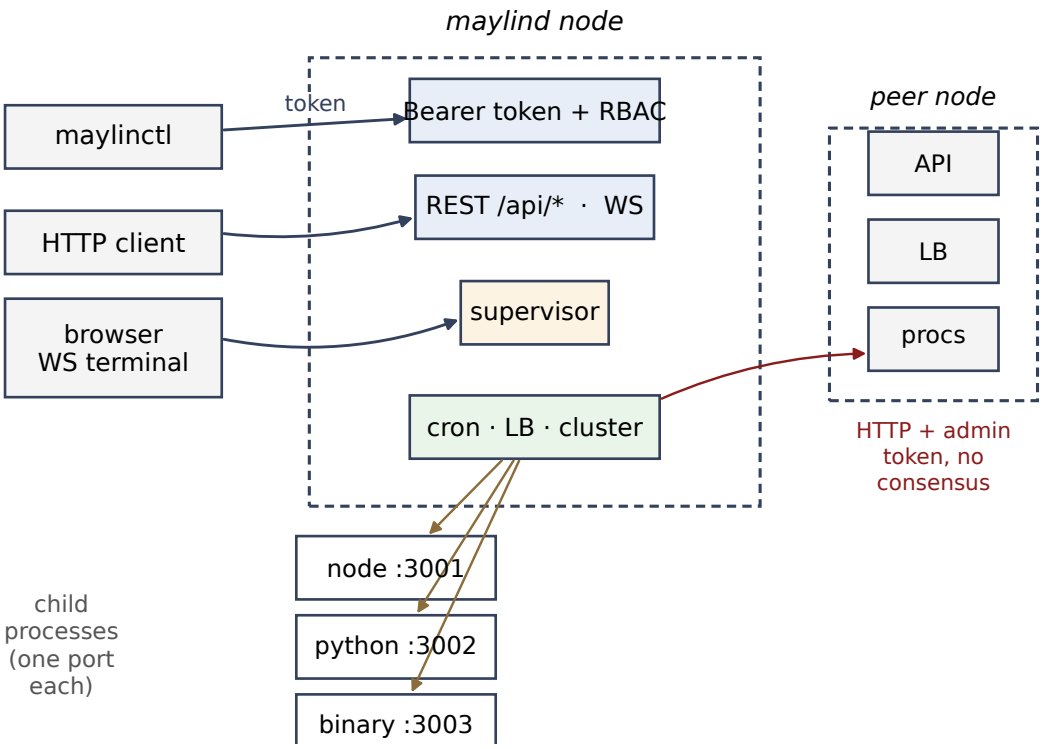


Figure 1: One maylin daemon per host. Services are ordinary children of the daemon; plugin templates turn a runtime into a uniform command line. The control plane is HTTP with a WebSocket attach endpoint; load-balancer listeners front replicas; peers are other maylin daemons reached over the same HTTP API with a shared token. Nothing in the diagram is a container.

3.2 Plugin templates and the uniform lifecycle

A plugin is eight fields: a name, an executable, optional arguments, an optional environment map, an optional script extension, and a few defaults. Substitution occurs on the executable and on each argument, replacing `{script}`, `{port}`, and `{name}`. The instance supplies the script path, the port, and its own name; the resulting vector is plugin arguments followed by instance arguments. The manager also injects `PORT` into the child's environment.

The important property is that substitution is textual on a single argument, never a shell expansion. A script path containing spaces, quotes, or a semicolon remains one argv element. The repository tests this directly: payloads such as `; touch pwned` are asserted to survive as a literal argument. This is a small thing, but it is the difference between a template system and a string concatenation bug, and it is the reason the plugin layer is safe to expose to operators who are not trusted to run arbitrary commands.

Three templates are shipped—`node`, `python`, and `binary`—and they differ only in the executable and the default arguments. Because the supervisor never inspects the runtime, a fourth template for a JVM or a .NET service requires no code change. The uniform lifecycle is therefore not an abstraction layer over runtimes; it is the observation that once a program is an argv vector with a port, nothing else about it matters to the supervisor.

3.3 The supervisor

Each instance is driven by one task that holds a start lock for its entire lifetime. The lock is the mechanism that prevents two supervisors for the same instance: a concurrent start blocks on it rather than launching a second child. It is also the source of a defect we measure in §5.6: the lock is held while the instance is running, so a second start on a live instance waits indefinitely rather than returning an error.

The task launches the child, records the start time, and enters a wait loop. On Windows the child's stdout and stderr are connected through an anonymous pipe created with `CreatePipe`, and completion is detected by polling `PeekNamedPipe` every 200 ms on a dedicated thread; the child is created with `CREATE_NO_WINDOW` so that launching a console program does not flash a window. On other platforms the wait loop uses the standard non-blocking wait. Output lines are read from the pipe, appended to an in-memory ring, and broadcast to WebSocket subscribers.

When the child exits, the supervisor computes one boolean:

```
code_ok = (exit_description == "exit code
                              0")
```

That is the entire crash detector. A signal death produces `signal: N`, which is not equal to the string, and is therefore a crash; a nonzero exit is a crash; only a literal zero is clean. The comparison is on a formatted string rather than an exit status, which is inelegant, but on the platform where it was written it is total: every termination path produces either an exit code or a signal description.

The policy then decides. `no` never restarts. `on-failure` restarts only when `code_ok` is false. `always` restarts regardless. If the previous run lasted longer than 120 seconds, the retry counter is reset to zero, so a service that crashes once a day is not eventually given up on. Otherwise the counter is incremented and compared with `max_retries`; if it exceeds the limit, a `giveup` event is emitted and the task returns, leaving the instance in `failed`. The backoff before the next launch is

```
Duration::from_secs((2 *
 retries).clamp(1, 30))
```

which is linear in the retry count, not exponential. The project's own documentation calls it exponential; the code is not, and §5.2 measures the linear sequence. Because the comparison happens after the increment and uses a strict inequality, a configuration of `max_retries = 3` permits four launches, not three. Both facts are reported here because a reader who trusts the documentation will size their retry budget wrongly.

Stopping is a cooperative protocol with a hard deadline. The handler sends a stop command to the task; the task writes a line to the child's stdin, waits `grace_secs` (minimum one second), and if the child is still alive calls kill. On Windows that is `TerminateProcess`, which does not run destructors and does not touch grandchildren. The handler waits up to 20 seconds for the state to become `stopped`, then returns success—without verifying that the child actually died. A stop that times out therefore returns the same message as a stop that worked.

Health checking is deliberately not part of the restart loop. A probe runs every `health_interval_secs` (minimum three seconds); three consecutive failures move the instance to `unhealthy` and emit an event, and a success moves it back to `running`. `unhealthy` is one of the states in which the instance is considered alive, so a start request is refused, but no restart is attempted. The design rationale is that a failing health probe usually means the service is up but degraded, and killing it would destroy diagnostic state; the cost is that health checking is advisory, and an operator who expects it to heal a hung service will be disappointed.

3.4 Cron and the supervisor

A schedule is five fields: a name, a five-field cron expression, a target instance, an action (`start`, `stop`, `restart`), and an enabled flag. At startup, one task is created per enabled schedule; each sleeps until the next matching local time, computed from the `cron` crate with a leading zero field added to make the expression six-field, then calls the same `process::start`, `stop`, or `restart` function that the HTTP handler calls.

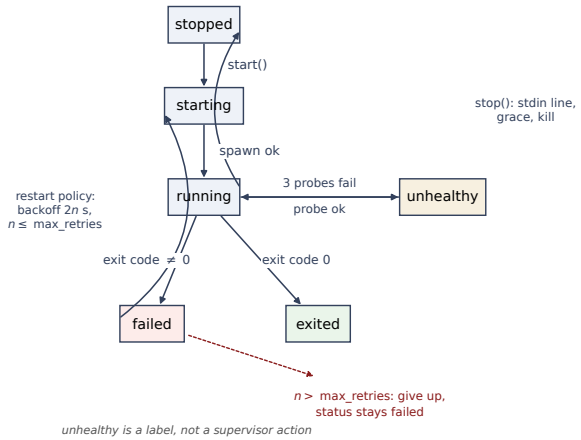


Figure 2: States and transitions of one instance. Solid arrows are supervisor actions; the dashed edge is the give-up path. Note that `unhealthy` has no outgoing edge to a restart: it is a label reported to operators, not an action taken by the supervisor.

Sharing the entry point is good for consistency and bad for semantics. The HTTP handler for restart is written for a human who has decided that now is a good moment; the cron task has decided nothing of the sort. There is no drain, no readiness gate, and no coordination with the load balancer, so a scheduled restart kills in-flight requests exactly as an interactive one does. There is also no single-flight across nodes: two hosts with the same schedule both fire, and if the target name resolves locally on both, both restart. The scheduler does not record the last run, does not catch up on missed windows, and does not jitter simultaneous schedules; for a handful of hosts this is tolerable, and it is stated here rather than discovered later.

The interaction is therefore simple to describe and unpleasant to rely on: *cron can restart a service in the middle of a request, and nothing in the system prevents it or waits for the request to finish.* What the load balancer does about it is measured in §5.4.

3.5 Roles and tokens

The token table has two parts. Tokens listed under `auth.tokens` are administrators. The table `auth.users` maps a token to a role name, and an unrecognized role name degrades to `viewer` rather than failing closed to denial—a deliberate choice so that a typo locks a user out of writes but not out of visibility. Roles are ordered `viewer` < `operator` < `admin`, and a middleware resolves the bearer token (or a `?token=` query parameter, §6.1) to a role before the handler runs.

Authorization itself is a single function of role, method, and path:

- any `GET` is allowed for `viewer`, except a path ending in `/terminal`;
- `POST` to `start`, `stop`, `restart`, or `reload` requires `operator`;
- every other `POST` and every `DELETE` requires `admin`.

There is no per-object scoping: an operator who may restart instance A may restart instance B. There is no notion of ownership, tenancy, or resource-level grants. This is a policy for a trusted team with a read-only audience, not a multi-tenant authorization system, and §6.1 enumerates exactly what it does and does not protect.

3.6 Cluster and load balancing

Clustering is a static peer list. Each node reads its own `maylin.toml` and knows the base URL and token of every peer. Status is an aggregation: the node issues a sequential `GET /api/status` to each peer with a five-second timeout and merges the answers for display. An action against a named instance is fanned out as one independent HTTP POST per peer, each authenticated with the cluster token (or, if absent, the first administrator token). There is no membership protocol, no heartbeat, no leader, no replicated log, and no conflict resolution. Two nodes may run an instance with the same name; they are two processes, and nothing in the system considers that a problem.

The load balancer is a separate listener. It keeps an ordered list of upstreams, a round-robin counter, and a health bit per upstream. Selection advances the counter modulo the number of upstreams and skips unhealthy entries; if all are unhealthy it falls back to the full list rather than failing. A request is forwarded with hop-by-hop headers stripped, the response body is buffered (up to 16 MiB) before being returned, and on a transport error—connection refused, reset, timeout—the next upstream is tried, up to the number of upstreams. An HTTP error status is not a transport error and is returned as-is. There is no per-request timeout of maylin's own, and the health checker holds the upstream table's write lock while probing sequentially, so requests can queue behind probes.

3.7 Terminal and command execution

Two endpoints deserve separate description because they are the sharpest edges in the system. `GET /api/instances/{name}/terminal` upgrades to a WebSocket, replays the last fifty log lines, and then forwards every text frame from the client to the instance's stdin, line by line, until `/quit` or disconnect. It is *not* a shell: it does not spawn a process, allocate a pseudo-terminal, or change the working directory. It is a remote stdin pipe with a log tail attached.

`POST /api/exec` is different: it spawns an arbitrary command through the platform shell, blocks a worker thread for up to `timeout_secs` (capped at 120), limits concurrency to four with a semaphore, and returns stdout, stderr, and the exit code. It is available to administrators only, and it is unabashedly remote code execution as the daemon's user.

4 Implementation

The daemon is built on tokio and axum 0.8, with reqwest for peer calls and the outbound side of the load balancer, tokio-tungstenite for the WebSocket test client, serde and toml for configuration, and clap for the command line. The source is divided into twelve modules; the supervision logic is the largest, followed by the HTTP API, configuration, and the load balancer.

State is shared through an `Arc<AppState>` containing a map from name to `Arc<Instance>`. An `Instance` carries its specification, an atomic status, an atomic desired flag, the restart counter, a broadcast channel for log lines, a ring buffer of recent lines, and the start lock. The control channel carries only two messages—stop and stdin—which is enough because every other operation is expressed as a state change followed by a wake-up.

The event log is append-only JSON lines, one object per event, with a timestamp, a kind, an instance name, a severity, and a message. It is the system's audit trail and also the measurement instrument used in §5: the intervals between `instance.exit` and `instance.start` events are the backoff, observed from outside the process rather than inferred from configuration. Authentication failures are counted and emitted as events, throttled to one per minute per source so that a brute-force attempt cannot fill the disk. There is no lockout: the throttle limits log volume, not attempts.

Configuration bootstrap is part of the binary. On first start, if `maylin.toml` is missing, the daemon writes a default configuration, three plugin files, two demo instances (one Node, one Python), and one schedule. This makes the first run self-explanatory and is also why the repository contains no `plugins/` directory: the templates are strings in the code, materialized on demand.

5 Evaluation

5.1 Method

All measurements come from driving the release-of-record debug binary over HTTP with real child processes, on one Windows host, with Python 3.13.14 as the service runtime. The harness (`scripts/measure_supervision.py`) is standard-library only: it writes instance and plugin TOML files into a temporary root, starts the daemon on a private port, waits for readiness, issues requests, reads the event log, and writes a JSON file (`scripts/results/supervision.json`). The paper's numbers are read from that file. Experiments are independent and each uses its own daemon and port, so a hang in one does not contaminate another.

The five experiments are: **A**, crash versus clean exit and the backoff sequence; **B**, stop latency against a child that ignores

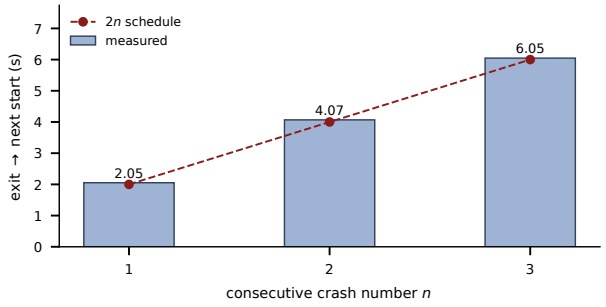


Figure 3: Measured intervals between launches of a crashing child (bars) against the $2n$ seconds the code computes (line). The intervals are linear; the documentation's description of exponential backoff is not what the binary does.

stdin; **C**, restart with requests in flight, with one and with two upstreams; **D**, an authorization matrix over every registered route and four principals; and **E**, a second start issued against a running instance. The test suites were run separately with `cargo test`.

5.2 Crash, clean exit, and backoff

Two instances were configured with `max_retries = 3`. The first exits with code 1 after 0.3 s; the second exits with code 0 after the same delay. After 22 seconds, the results were:

	crashing	clean
final status	failed	exited
last exit	exit code 1	exit code 0
launches	4	1
exit events	4	1
give-up events	1	0

The crashing child was launched four times for a configured limit of three, confirming the off-by-one described in §3.3. The clean child was launched once and left in `exited`, which is the correct behaviour for a one-shot job and the reason `on-failure` is the default policy. The measured gaps between consecutive launches were 2,053, 4,068, and 6,047 milliseconds (Figure 3), matching $2n$ seconds with roughly 50 ms of process-spawn overhead. A reader who expected exponential backoff would predict 2, 4, 8; the third gap is 6, not 8. The difference matters at the tail: linear backoff reaches the 30-second clamp after fifteen retries, while exponential backoff reaches it after five.

Two failure modes are visible in this experiment and worth naming. First, a one-shot job that exits nonzero for a legitimate reason—"no work to do" is often code 2—is indistinguishable from a crash under `on-failure`. Second, the give-up path leaves the instance in `failed`, which is a terminal state until an operator intervenes; there is no automatic recovery after a cool-down. Both are defensible choices, but they are choices, and an operator who expects a supervisor to keep trying forever will be surprised.

5.3 Stopping a child that does not cooperate

An instance was started that records a heartbeat but never reads stdin; its grace period was one second. The stop request returned HTTP 200 after 1.207 seconds, and the status immediately afterwards was `stopped`. A second stop returned HTTP 500 with the message that the instance is not running.

The latency is what the design predicts: one second of grace plus the polling interval plus teardown. The 500 is a smaller issue with a diagnostic cost: stop is not idempotent, so a retry after a timeout—the natural response to a timeout—reports failure even though the desired state holds. The more serious point is that the handler's success is not evidence of death. It waits up to 20 seconds and then returns success regardless; in this experiment the child did die at the deadline, but a child that survives `TerminateProcess` (a zombie on some platforms, a process stuck in an uninterruptible call) would produce the same 200. The event log is the only reliable witness, and the API does not say so.

5.4 Restart with requests in flight

This is the experiment that answers the cron question directly. A Python service records a side effect to a file, then sleeps long enough that the request is still in flight when a restart arrives. The restart is issued through the HTTP endpoint, which calls the same function the scheduler calls; the timing is therefore a faithful proxy for a cron-triggered restart, with the advantage that it is deterministic.

With a single upstream, four concurrent requests were issued and a restart was triggered 0.2 s later. The result was four 502 responses, four recorded side effects, and latencies of 1.43–1.49 s. Every client saw an error; every request had nonetheless reached the service and been executed. This is the worst combination: the work happened, and the caller does not know it.

With two upstreams behind the balancer, the same four concurrent requests all returned 200, with latencies of 3.03, 3.01, 4.46, and 5.51 seconds. The service recorded six executions for four requests, so at least two side effects were duplicated. The duplication is explained by the retry rule: a transport error after the request has been written causes the balancer to replay the request against the next upstream, and the first upstream may already have executed it. The multi-second latencies are consistent with requests waiting while the health checker holds the upstream table and probes sequentially; we did not instrument the lock, so this is an explanation rather than a measurement.

	one upstream	two upstreams
requests	4	4
HTTP 200	0	4
HTTP 502	4	0
latency range (s)	1.43–1.49	3.01–5.51
side effects	4	6

The design lesson is not that the balancer is broken. It is that *a supervisor's restart and a proxy's retry are the same event from the service's point of view*, and neither layer has

Endpoint	adm	op	view	none
GET /api/status	200	200	200	401
GET /api/instances	200	200	200	401
POST /api/instances	200	403	403	401
GET /api/instances/{n}	200	200	200	401
DELETE /api/instances/ghost	500	403	403	401
POST /{n}/start (running)	T	T	403	401
POST /{n}/stop	200	500	403	401
POST /{n}/restart	200	200	403	401
GET /{n}/logs	200	200	200	401
GET /{n}/terminal	400	400	403	401
POST /api/exec	200	403	403	401
GET /api/plugins	200	200	200	401
GET /api/schedules	200	200	200	401
POST /api/reload	200	200	403	401
GET /api/lb/status	200	200	200	401
GET /api/cluster/status	200	200	200	401
POST /api/cluster/action	200	403	403	401
GET /api/events	200	200	200	401

Table 1: Measured status codes for every registered method–path pair and four principals. `T` = client timeout after 8 s. `{n}` abbreviates `/api/instances/{name}`. Three cells are artifacts of execution order rather than authorization: operator `stop` returned 500 because the preceding admin stop had already stopped the instance; `DELETE ghost` returned 500 because deleting a missing instance is an error rather than a 404; `terminal` returned 400 because the request carried no WebSocket upgrade header.

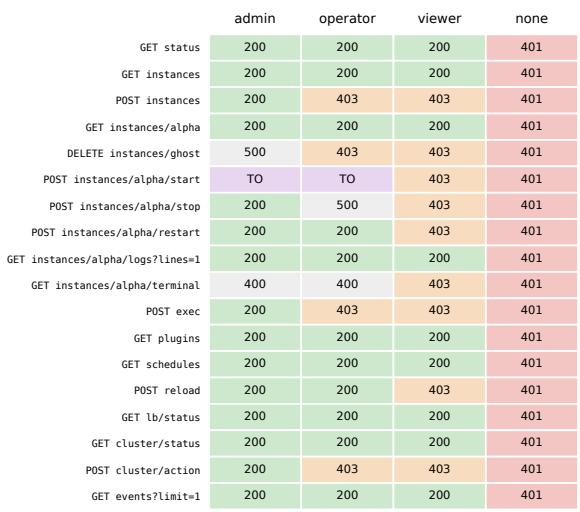


Figure 4: The authorization matrix as a heatmap: rows are endpoints, columns are principals, and each cell is the measured HTTP status. The vertical boundary between column three and four is the authentication check; the staircase inside the left three columns is the role heuristic of §3.5.

a concept of request identity. Maylin provides at-most-once delivery only when there is one upstream and no retry; with two upstreams it provides at-least-once for transport failures, which means non-idempotent endpoints can be executed twice. A production system would need request identifiers and a deduplication window, and maylin does not have them.

5.5 Authorization matrix

All 18 method–path pairs were requested with four principals: an administrator token, an operator token, a viewer token, and no token. Each request had an eight-second client timeout so that a hang would be recorded rather than stalling the experiment. The full matrix is Table 1; the heatmap in Figure 4 is the same data.

Three results stand out. First, anonymous access is uniformly rejected with 401, including the routes that are open to viewers; the middleware runs before every handler. Second, the role boundary is exactly where the heuristic puts it: viewers are refused on the terminal endpoint and on every mutating method, operators are refused on instance creation, deletion, exec, and cluster actions. Third, the failures are not authorization failures but hygiene failures: a missing object produces 500 rather than 404, a start on a running instance hangs, and a terminal request without an upgrade header produces 400 rather than a clear message. None of these leaks data, but all of them make automation harder, and a client cannot distinguish "not allowed" from "not found" from "broken" by anything other than the status class.

5.6 A second start on a running instance

Because the lifecycle task holds the start lock for its whole life, a start request against a running instance waits on the lock rather than observing that the instance is alive and returning an error. In the experiment, the request did not complete within the eight-second client timeout (8.01 s measured), while the instance remained `running`. After the client gave up, the instance was stopped; it was not restarted by the abandoned request.

Two things follow. Operationally, the request is a connection and a task held open for as long as the instance runs, which is a cheap denial of service for anyone who can reach the endpoint (an operator, in the current model). Diagnostically, the intended check—"if the current status is alive, return an error"—exists in the handler but is placed after the lock acquisition, so it can never fire while a supervisor is alive. The fix is to test the status before acquiring the lock and again after; the experiment is included here because the defect is invisible to a casual test and obvious under a matrix that includes a start against a running service.

5.7 Test suites

The repository contains 33 test functions. The 28 unit tests pass in 0.04 s and cover configuration parsing, plugin substitution and injection, the authorization heuristic, and the state machine's definitions. The four authentication and injection integration tests pass in 5.42 s; they start a real daemon and assert that SQL injection, XSS, a token in the query string, and a path traversal are rejected or handled without side effects. The end-to-end test is a single function with eleven steps—readiness, two instances running, direct connect, round-robin,

failover, recovery, exec, authentication rejection, event system, persistence, cleanup—and it passed in 17.52 s on the third attempt.

The first two attempts failed, at different steps, and the failures are a finding rather than noise. Attempt one hung at the failover step for 121 seconds. Attempt two failed at the direct-connect step after 10.93 seconds because ports 13301 and 13302 were still bound by child processes from the previous run: the harness's cleanup calls `kill` on the daemon, which on Windows is `TerminateProcess`, and terminating the parent does not terminate the grandchildren. The third attempt passed only after those orphans were killed by hand. A supervisor whose test harness leaks the very processes it supervises is telling you something about the supervision boundary: on Windows, the process tree is not a unit, and "kill the manager" does not mean "stop the services".

5.8 Threats to validity

The measurements come from one host, one operating system, and four requests per condition. The four-request sample is enough to establish that the failure happens and to bound the latency coarsely, not enough to estimate a distribution. Windows is the platform where the pipe implementation is most unusual, so the Linux path is less tested by these experiments, not more. The daemon was run in debug mode, so absolute latencies are pessimistic; the comparisons between conditions are unaffected. The authorization matrix uses one instance state taken at one moment, and three cells are artifacts of that state, as noted in the table. No memory, CPU, or long-run soak measurement was made, and no experiment exercised more than two upstreams.

6 Security and failure analysis

6.1 Enforcement points

Authorization is enforced at exactly one place: a middleware that resolves a token to a role and a pure function that maps (role, method, path) to a boolean. That is a good shape—one place to audit—and the matrix shows it works for the policies it implements. The limits are equally clear.

The predicate is a suffix heuristic, not a route table. It decides that a path is a terminal if it ends with `/terminal`, that a POST is a lifecycle action if it ends with `start`, `stop`, `restart`, or `reload`, and that everything else mutating is administrative. A path with a trailing slash, `/terminal/`, does not match the suffix and is therefore treated as a plain GET allowed to viewers; the unit tests document this and the router then returns 404, so there is no bypass today. But the safety of the rule depends on the router's normal form matching the predicate's normal form, which is an invariant no test enforces for future routes.

Tokens may be supplied as `?token=` in the query string. This is convenient for command-line use and for WebSocket clients that cannot set headers, and it means tokens appear in access logs, proxy logs, browser history, and shell history. Bearer tokens in headers are the recommended form (Jones and Hardt, 2012); the query fallback should be restricted to the terminal upgrade or removed.

The role model is flat and global. There is no per-instance grant, so an operator trusted with a staging worker is trusted with the database. There is no separation between reading logs (which may contain secrets) and reading status; both are viewer-level. There is no lockout after repeated failures, only a one-per-minute event throttle. And the "unknown role becomes viewer" rule fails open in the sense that a mistyped role still grants read access to everything, including logs and the event stream.

6.2 Terminal and exec

The WebSocket terminal is frequently described as remote code execution, and the precise truth is more interesting. It does not start a shell; it writes lines to the stdin of a service that is already running. For a service that treats stdin as a command channel—a Python REPL, a Node REPL, an interactive CLI, or the shipped `binary` template pointed at an interpreter—that is remote code execution as the service's user, with the service's environment and file descriptors, and it is available to operators, not only administrators. For a service that ignores stdin, it is a log tail with a write-only appendix. The blast radius is bounded by which service is attached: the endpoint cannot choose a process, cannot change the working directory, and cannot spawn anything. It can, however, read the last fifty log lines before the operator types anything, and logs routinely contain paths, hostnames, and occasionally credentials.

`/api/exec` is the blunter instrument. It runs an arbitrary command through the platform shell, as the daemon's user, with a 120-second cap and four-way concurrency limit. It is administrator-only, which in this model means anyone holding a token from `auth.tokens`—typically the same token used for cluster fan-out. There is no allow-list, no sandbox, no working-directory restriction, and no audit beyond an event containing the first 120 characters of the command. An administrator compromised is the host compromised; this is stated in the documentation and should be stated in any deployment review.

6.3 No isolation

This is the property that distinguishes maylin from every container system, and it deserves to be in the security section rather than only in limitations. A service runs as the daemon's user, with the daemon's environment, in a working directory chosen by configuration, with full access to the host filesystem subject only to the operating system's permissions. It can read the daemon's configuration, including other services' tokens and environment variables. It can bind any port, including the control plane's. It can signal or kill other processes owned by the same user, including the daemon itself. It can exhaust memory, file descriptors, and disk without a cgroup to stop it.

None of this is accidental; it is the price of direct host access, and it is the correct price for the target deployment. But it

means that maylin's security boundary is the operating-system user, not the service. Two services that must not read each other's files require two users, and maylin has no notion of a per-service user. A compromised service is a compromised host, and the RBAC layer protects the API, not the services from each other.

The same absence explains the reproducibility gap. A container image pins the interpreter, the libraries, and the filesystem layout. Maylin pins a command line. If the host's Python is upgraded, every Python service changes behaviour at its next restart, and the event log will show a restart with no configuration change—the classic dependency-drift failure. There is no lockfile, no environment hash, and no record of what the interpreter version was when the service last worked.

6.4 Cluster membership and split-brain

There is no membership protocol, so the usual split-brain question does not arise in the usual form. Each node reads a static list of peers and treats each as reachable or not; there is no quorum, no term, no leader, and no replicated state to diverge. The consensus literature (Lamport, 1978; Ongaro and Ousterhout, 2014) is not applicable because there is nothing to agree on.

What exists instead is a weaker and more insidious failure: two nodes can each be running an instance with the same name, each believing it is authoritative for its own copy, and the cluster status view will show both as running. Nothing detects the duplication, nothing arbitrates, and a load balancer configured with both will happily send traffic to both. If the two copies share a database, the duplicate is a correctness bug; if they share a port on a shared host, one of them failed to bind and the operator sees a crash loop. The static peer list makes this more likely, not less: a node added to two peer lists, or a configuration copied to a new host, produces a duplicate without any protocol violation.

The fan-out path has a concrete defect. For a named target, the node issues one POST per peer and each peer acts on its local instance of that name, which is the intended semantics. For the target `all`, the code first fetches each peer's `/api/instances` and then attempts to treat the response as a JSON array. The endpoint returns an object with an `instances` field, so the parse yields nothing and the peer is skipped silently. The local node is not included in the fan-out either. The request returns 200. In other words, the one operation an operator is most likely to reach for in an incident—"stop everything everywhere"—reports success while doing nothing on the peers. This is not a consensus failure; it is a missing test, and it is reported here because it is exactly the kind of gap that a feature list hides.

6.5 Supervisor failure modes

The supervisor has five behaviours that an operator should know before trusting it. A crash loop with `always` restarts until the retry limit, then gives up and stays `failed`; there is no cool-down and no escalation. `max_retries` $= k$ means $k + 1$ launches. A stop that exceeds the 20-second wait returns success without verifying death. A health probe that fails three times marks the instance `unhealthy` but never restarts it, and because `unhealthy` counts as alive, a start request is refused—the instance is stuck until an operator stops it. And on Windows, kill is `TerminateProcess`, which leaves grandchildren running; a service that spawns helpers will leak them across restarts. That last behaviour is not hypothetical: it is what made the first two end-to-end runs fail.

6.6 Load-balancer failure modes

The balancer is a correct round-robin for the common case and has four sharp edges. It retries only on transport errors, so an upstream that returns 500 is never bypassed; a broken service that accepts connections and fails requests will receive its full share of traffic indefinitely. It buffers the entire response body up to 16 MiB before forwarding, so a streaming endpoint is defeated and a large download is held in memory. It sets no request timeout, so a hung upstream holds a connection until the client gives up. And its health check holds the upstream table's write lock across sequential probes with three-second timeouts, so with many upstreams a probe round can block selection for seconds; the 3.0–5.5 s latencies in §5.4 are consistent with this, though not instrumented.

7 Limitations

The evaluation is one host, one platform, and small samples; §5 states the threats in detail. Beyond the measurements, four limitations are structural.

No isolation and no reproducibility. As described in §6, the security boundary is the OS user and the reproducibility boundary is the host's disk. Maylin cannot run two services that must not see each other, and it cannot reproduce a deployment from a manifest.

No consensus, no membership, no reconciliation. The cluster is a static list of peers with per-node authority. Configuration drift between nodes is undetected, duplicate instance names are undetected, and there is no mechanism to converge a node to a desired state. Adding a node is a configuration edit on every node that should know about it.

The scheduler is a timer, not a workflow engine. It fires one action at one cron time, with no dependencies, no retries, no catch-up, no jitter, and no coordination with the supervisor beyond calling the same function. It is suitable for "restart the worker at 03:00" and not for "run this after that succeeds".

The API is a control surface, not a contract. There is no versioning, no OpenAPI description, no idempotency keys, and no pagination on the event stream beyond a limit parameter. The 500-on-missing-object and the hanging start are symptoms of a surface designed for a UI rather than for automation.

8 Related work

Process supervision. systemd (Poettering, 2024) is the reference for lifecycle management on Linux: units, cgroup accounting, restart policies, watchdogs, and a dependency graph. supervisord (Supervisor Development Team, 2024) is the closest in spirit—a configuration file, a set of child processes, an HTTP interface—and is the system maylin most resembles, minus roles, clustering, and the load balancer. runit and s6 take the opposite approach, encoding supervision in small directory-driven programs with no control plane at all. Erlang/OTP (Armstrong, 2003) contributes the supervision tree, restart intensity, and the idea that a supervisor's own failure is part of the design; maylin has a flat set of supervisors with no hierarchy. The crash-only philosophy (Candea and Fox, 2003) argues that recovery should be restart rather than repair, which is maylin's default posture, though its health check deliberately declines to restart.

Container orchestration. Docker (Merkel, 2014), Compose (Docker, Inc., 2024), Kubernetes (The Kubernetes Authors, 2024), and Borg (Burns et al., 2016) solve isolation, packaging, scheduling, and reconciliation together. Maylin solves none of those and should not be compared feature by feature; the honest comparison is that a Compose file and a maylin configuration describe a similar topology, and Compose buys isolation and reproducibility at the cost of a runtime and an image pipeline. The namespaces (Biederman, 2006) and cgroups (Menage, 2007) literature explains what is given up; the OCI specification (Open Container Initiative, 2024) explains what a container runtime must implement. Sandboxing tools such as seccomp (Linux man-pages project, 2024b), Landlock (Linux man-pages project, 2024a), and bubblewrap (bubblewrap contributors, 2024) show that isolation can be added without a full container stack, which is a plausible direction maylin has not taken.

Access control and distribution. Role-based access control (Sandhu et al., 1996) and the principle of least privilege (Saltzer and Schroeder, 1975) are the frame for §6.1; maylin implements a coarse role lattice and no object-level policy. Bearer token usage follows RFC 6750 (Jones and Hardt, 2012) in form but not in the recommendation against query parameters. The cluster's lack of consensus is conspicuous against Raft (Ongaro and Ousterhout, 2014) and the classic distributed-systems results on partial failure (Lamport, 1978, 1998; Gilbert and Lynch, 2002); maylin's position is that a static, small deployment can avoid the problem by not having shared state, and §6.4 shows what that avoidance costs. Gray failure (Huang et al., 2017) and the SRE literature (Beyer et al., 2016) describe the operational failure modes—health checks that pass while the service is degraded, retries that amplify load—that the measurements in §5.4 reproduce in miniature. Release It! (Nygard, 2018) names the circuit breaker and bulkhead patterns that the load balancer does not implement.

9 Conclusion

Maylin is a service cluster manager for people who want their services to be processes. Its plugin templates turn a runtime into an argv vector; its supervisor gives that vector six states, three policies, a linear backoff, and an event log; its cron scheduler and its HTTP API call the same functions; its token table gives viewers, operators, and administrators different verbs; its peers are other copies of itself. The measurements show the parts working and the seams showing: four launches for a limit of three, 2/4/6-second gaps where the documentation promised exponential growth, a 1.21-second stop that reports success without proof of death, four lost requests against one upstream and two duplicated side effects against two, and 18 endpoints that reject anonymous callers while returning 500 or hanging when the object or the state is wrong.

The larger claim is about scope. A supervisor is not a scheduler, a proxy is not a transaction manager, and a static peer list is not a cluster membership protocol. Maylin is useful precisely because it stops at those boundaries and because its operator can read all of it—4,871 lines, one binary, five TOML files per host. It gives up isolation, reproducibility, and consensus, and the paper has tried to be explicit about what those give-ups cost. For the deployment it targets, a handful of hosts and a team that already owns them, that is a reasonable trade. For anything larger, the conventional stack is conventional for good reasons.

Availability. The artifact package contains the sources, the measurement harness, the JSON results, and the scripts that regenerate every figure. The repository is at commit `3c285fe`; the paper's figures are produced by `scripts/make_figures.py` from `scripts/results/supervision.json`.

References

J. Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, Royal Institute of Technology (KTH), Stockholm, 2003.

B. Beyer, C. Jones, J. Petoff, and N. R. Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016.

E. W. Biederman. Multiple instances of the global Linux namespaces. In *Proceedings of the Linux Symposium*, 2006.

bubblewrap contributors. bubblewrap: Unprivileged sandboxing tool. https://github.com/containers/bubblewrap, 2024.

B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14(1):70–93, 2016.

G. Candea and A. Fox. Crash-only software. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, 2003.

Docker, Inc. Docker compose overview. https://docs.docker.com/compose/, 2024.

S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

P. Huang, C. Guo, L. Zhou, et al. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*, 2017.

M. Jones and D. Hardt. The OAuth 2.0 authorization framework: Bearer token usage. RFC 6750, IETF, 2012.

L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

Linux man-pages project. `landlock(7)`: Linux manual page. https://man7.org/linux/man-pages/man7/landlock.7.html, 2024a.

Linux man-pages project. `seccomp(2)`: Linux manual page. https://man7.org/linux/man-pages/man2/seccomp.2.html, 2024b.

P. Menage. Adding generic process containers to the Linux kernel. In *Proceedings of the Linux Symposium*, 2007.

D. Merkel. Docker: Lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014.

M. T. Nygard. *Release It! Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2nd edition, 2018.

D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC '14)*, pages 305–319, 2014.

Open Container Initiative. Open container initiative runtime specification. https://github.com/opencontainers/runtime-spec, 2024.

L. Poettering. systemd system and service manager. https://systemd.io/, 2024. Manual pages `systemd.service(5)`, `systemd.exec(5)`, `systemd.unit(5)`.

J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

Supervisor Development Team. Supervisor: A process control system. http://supervisord.org/, 2024. Documentation, including `[program:x]` sections and `autorestart` semantics.

The Kubernetes Authors. Kubernetes documentation: Liveness, readiness and startup probes. https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/, 2024.

A. Wiggins. The twelve-factor app. https://12factor.net/, 2011.