

NewWave — Technical Architecture

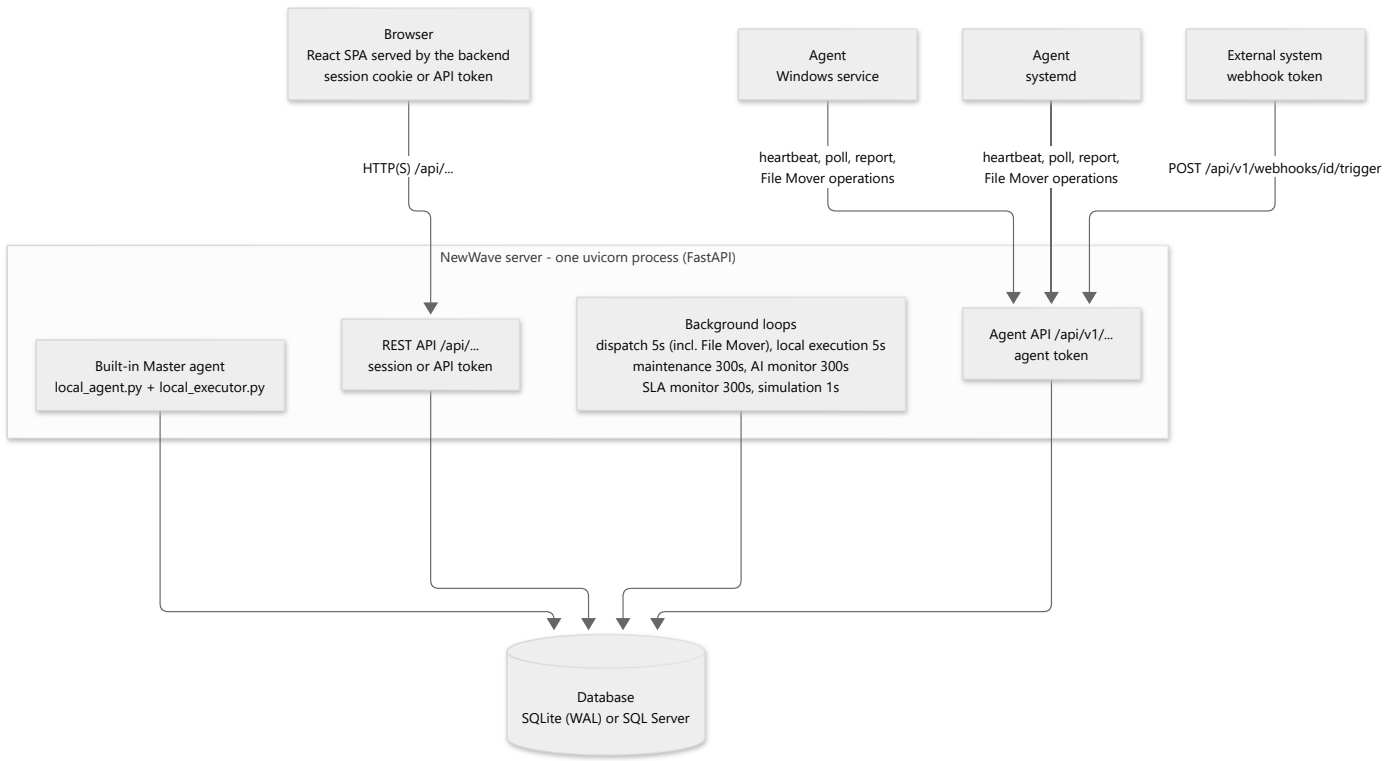
NewWave is a workload-automation (job scheduling) platform: a web console and scheduler server, plus **agents** that execute jobs on remote hosts.

This document describes how the system is put together, with particular detail on the **agent** and the protocol between agent and server. It reflects the code at repository revision `7f3571e` plus the working tree of 2026-09-24 (File Mover, webhook variables, operational fixes); the agent is a separate repository (`newwave_agent`) at build `nw-agent-20260924-083159` . File references are relative to `backend/app/` (server), `src/` (frontend) or the agent repository root, as marked.

Contents

1. [System overview](#)
 2. [Technology stack](#)
 3. [Server architecture](#)
 4. [Data model](#)
 5. [Scheduling engine](#)
 6. [Agents](#)
 7. [Agent protocol](#)
 8. [Run lifecycle end to end](#)
 9. [High availability](#)
 10. [Plugins, job types and integrations](#)
 11. [Security](#)
 12. [Frontend](#)
 13. [Deployment and operations](#)
 14. [Known limitations](#)
-

1. System overview



Three properties shape most of the design:

- **Pull, not push.** The server never connects to an agent. Agents poll for work over HTTP, so agents need no inbound ports and can sit behind NAT or firewalls. See §7.
- **The database is the coordination layer.** Runs move through a status machine held in the `job_runs` table. Claiming work, reserving capacity and holding the HA lease are all conditional `UPDATE`s (or row locks), so several threads, processes or instances can act on one database safely.
- **One deployable.** The backend serves the built frontend (`dist/`) from the same port, so a full install is one service plus a database.

2. Technology stack

Layer	Technology
Server	Python 3.11+, FastAPI, Starlette (sessions, CORS), uvicorn
Persistence	SQLAlchemy 2.x ORM; Alembic (175 migrations); SQLite (default, WAL mode) or SQL Server via pyodbc
Frontend	React 19, TypeScript, Vite, React Router 7, TanStack Query / Table / Virtual, @xyflow/react + @dagrejs/dagre (flow and workflow views), CSS modules
Agent	Python 3.11+, asyncio, httpx, pydantic-settings; pywin32 (Windows service); paramiko and pyftplib (optional file-transfer server)

Layer	Technology
Licensing	Ed25519 signature verification (<code>licensing.py</code>); licenses are issued by a separate private tool
AI (optional)	Anthropic Python SDK; model and API key configured under Administration → AI Monitor
Windows hosting	NSSM service running <code>uvicorn app.main:app</code> on port 8010
Standalone edition	PyInstaller one-file executable (<code>deploy/idruns.spec</code> , <code>deploy/launcher.py</code>) for Windows and Linux

`SQLite` is a single-process development/test option. `SQL Server` is the production target and is required for HA, because HA means more than one server process sharing one database.

3. Server architecture

3.1 Process model

`app.main:app` is a single FastAPI application. Its `lifespan()` (`main.py`):

1. loads plugin adapters (`load_plugins()`);
2. creates or refreshes the built-in **Master Agent** row (`_ensure_master_agent`) and the singleton `SchedulerConfig`;
3. seeds demo data on a fresh install, and demo mode if `DEMO_MODE=true`;
4. registers the local-cancel hook so `scheduler.cancel_run()` can stop locally executing runs directly;
5. starts the background loops below;
6. on shutdown cancels the loops and flushes the audit writer.

3.2 Background loops

Loop	Cadence	Work
<code>_dispatch_loop</code>	<code>DISPATCH_TICK_SECONDS</code> , default 5 s	<code>run_dispatch_tick</code> , <code>group_dispatch_tick</code> , File Mover coordinator tick (<code>file_mover_engine.tick</code>), <code>process_terminate_requests</code> , <code>enforce_max_runtime</code> , <code>reap_stalled_runs</code>
<code>_local_execution_loop</code>	5 s	Claims READY runs for the Master agent and executes them in-process
<code>_daily_maintenance_loop</code>	wakes every 300 s	Every wake: compiles the schedule (idempotent) and sweeps rotated service logs; once per day: carry-over, purge old runs and audit log, recompute calendar forecasts
<code>_ai_monitor_loop</code>	wakes every 300 s	Optional advisory findings and urgent critical-finding emails; runs only when due

Loop	Cadence	Work
<code>_sla_monitor_loop</code>	wakes every 300 s	<code>sla_monitor.check_sla_breaches</code> : raises at-risk and missed-deadline alerts (§5.8)
<code>_simulation_loop</code>	1 s	Advances explicitly started virtual-time simulation sessions; uses separate tables and cannot affect production runs

Each loop records attempts, successes and consecutive failures, and backs off with `_retry_delay` after a failure. The dispatch and maintenance ticks run in a worker thread (`run_in_executor`) so a slow tick cannot stall the event loop that also serves agent polls and UI requests. The local execution tick is the exception: it must run natively on the event loop because process supervision is `asyncio` .

Every tick first checks whether it is allowed to act:

```

HA enabled and lease not held → STANDBY (do nothing)
Master connection stopped    → PAUSED_MASTER_STOPPED (no new READY runs;
                               still manage in-flight runs)
License does not permit      → PAUSED_LICENSE_EXPIRED (same behaviour)
otherwise                    → full tick

```

In the paused states the File Mover tick still runs with `start_new=False` : workflows already in progress keep advancing and recovering, but no new one starts.

3.3 Request pipeline

Middleware order, outermost first: `SessionMiddleware` → `CORSMiddleware` → `auth_gate_middleware` → `access-log middleware` → `route`.

`auth_gate_middleware` requires a logged-in user for every `/api/` path except the prefixes `/api/v1/` (agent protocol, File Mover agent API and webhook triggers, each with its own token check) and `/api/auth/` . It then applies role rules by HTTP method and path:

Role	Capability
ADMIN	Everything, including Administration routes (<code>require_admin</code>)
USER	Read and write within the workgroups the user belongs to
SCHEDULER	Operate on Job Activity (rerun, cancel, hold, run now, overrides); blocked from creating, editing or deleting definitions
INQUIRY	Read-only; every non-GET request is rejected
DEMO	Like <code>USER</code> , confined to the <code>demo</code> workgroup, and mutations limited to an allowlist of definition paths plus run-now, rerun and cancel

Two further mechanisms keep the server responsive under load:

- **Heavy-read limiter** (`read_limits.py`): the few large read endpoints (job runs, job definitions, calendars, dependency batches) run on the thread pool but only `NW_HEAVY_READ_CONCURRENCY` (default 2) at a time, waiting on the event loop rather than in pool threads. Identical concurrent requests share one result for `NW_SHARED_READ_TTL` seconds (default 3). This exists because `pyodbc` contends for the GIL when many threads fetch large result sets in one process.
- **Asynchronous audit writer** (`audit_models.py`): audit rows are captured in `after_commit` and written by a dedicated thread through a bounded queue, so auditing does not extend request transactions.

3.4 Configuration

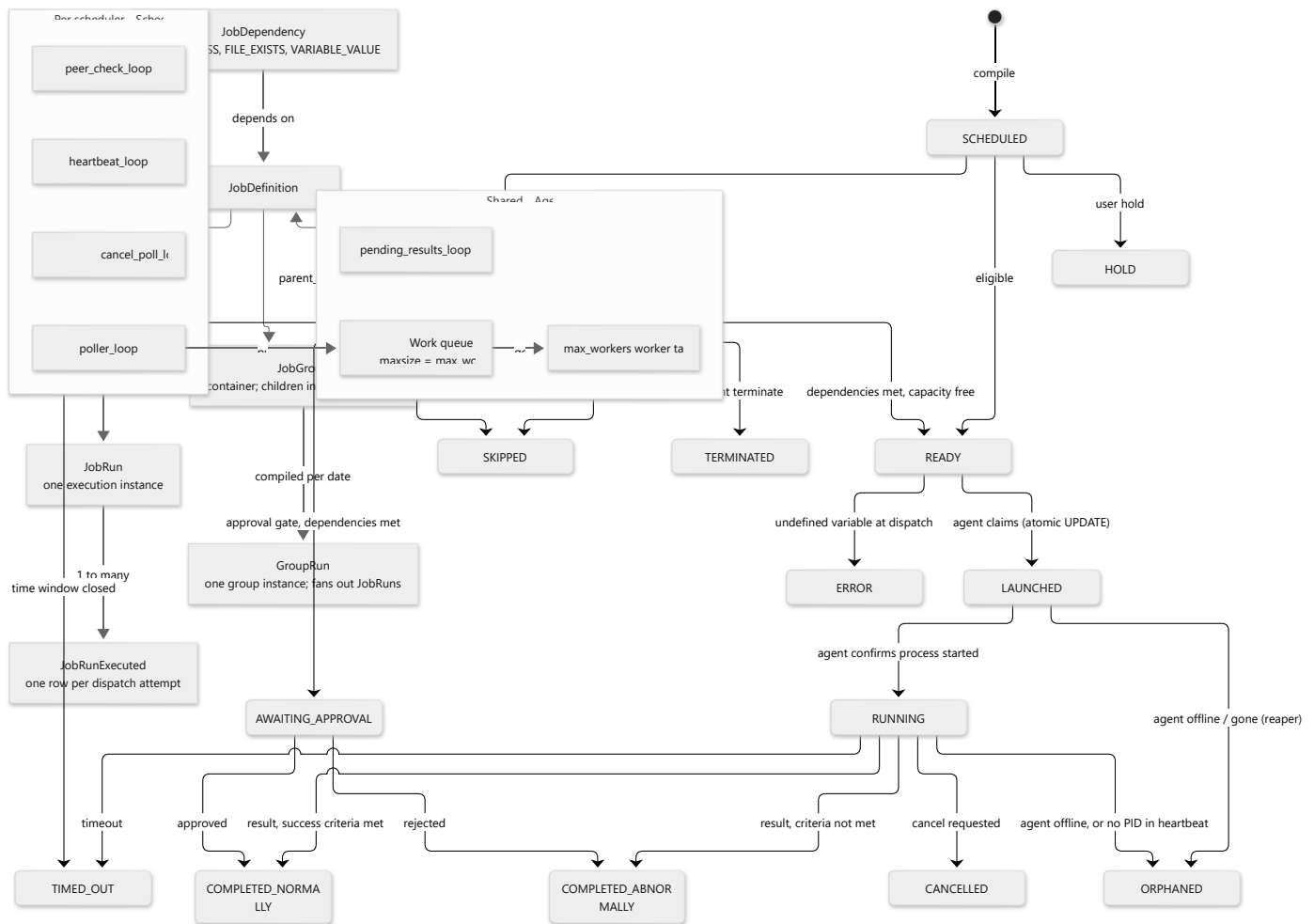
Environment variables, loaded from `backend/.env` (`settings.py` , `main.py` , `db.py`). The important ones:

Variable	Purpose
<code>SECRET_KEY</code>	Signs session cookies. Required ; the app refuses to start without it.
<code>DATABASE_URL</code>	Defaults to <code>sqlite:///./scheduler.db</code> . SQL Server: <code>mssql+pyodbc://...</code>
<code>SESSION_TIMEOUT_MINUTES</code>	Session lifetime, default 15
<code>SESSION_COOKIE_SECURE</code>	Mark the session cookie HTTPS-only
<code>CORS_ALLOWED_ORIGINS</code>	Allowed browser origins
<code>AGENT_AUTH_REQUIRED</code>	Enforce agent bearer tokens (default off; see §11)
<code>NW_LOCAL_EXECUTION_MAX_CONCURRENCY</code>	Cap on concurrent Master-agent runs, default 8
<code>NW_SCHEDULER_INSTANCE_NAME</code>	Human label for this instance in HA
<code>DEMO_MODE</code> , <code>PRODUCT_NAME</code> , <code>PRODUCT_TAGLINE</code>	Demo sandbox and branding
<code>NW_FILE_MOVER_RELAY_ROOT</code> , <code>NW_FILE_MOVER_RELAY_QUOTA</code>	File Mover relay directory (default <code>backend/file-mover-relay</code>) and reservation cap (default 50 GiB)
<code>LOG_DIR</code> , <code>LOG_FILE</code>	Application log location (default <code>backend/logs/backend.log</code>)
<code>SERVICE_LOG_DIR</code> , <code>SERVICE_LOG_MAX_FILES</code>	Where NSSM writes <code>service.out.log</code> / <code>service.err.log</code> (default repo-root <code>logs/</code>) and how many rotated copies to keep (default 20)

Operational settings that admins change at run time live in the singleton `SchedulerConfig` row (Administration → System Settings), for example `compile_days_ahead` (3), `agent_heartbeat_timeout_seconds` (120), `ha_enabled` , `ha_lease_timeout_seconds` (30), `peer_base_url` , retention periods, and feature switches (Job Iteration, Schedule Simulator, Version History, Import and Export TIDAL, Enable AI).

4. Data model

All models share one SQLAlchemy `Base` (`models.py`) and are written to be portable between SQLite and SQL Server (no database-specific types, no database-generated sequences).



Area	Tables (module)
Definitions	JobDefinition, Job, JobGroup (joined-table inheritance), JobDependency, RuntimeUser, PluginConnection (models.py, jobrun_models.py)
Runs	JobRun, GroupRun, JobRunExecuted, JobRunDependencyOverride, FileCheckCache, RunIdSequence, BatchIdSequence, ClearedScheduleDate (jobrun_models.py)
Scheduler state	SchedulerConfig (singleton), SchedulerLease (singleton, HA)
Agents	Agent, AgentList, AgentListMember (agent_models.py, agentlist_models.py)
Variables	Variable (global), GroupVariable, JobVariable
Calendars	Calendar, FiscalCalendar
Events and actions	Action, JobEventType, FileEventType, VariableEventType, SystemEventType, their firing and action-step tables (event_models.py)
Access	InteractiveUser, WorkGroup, ApiToken, LdapConfig (auth_models.py, workgroup_models.py)
Observability	AuditLogEntry, Alert, UserMessage

Area	Tables (module)
Scheduling controls	<code>ResourcePool</code> , <code>JobResourceRequirement</code> (<code>resource_models.py</code>); <code>BranchRule</code> (<code>branching_models.py</code>); <code>Job.priority</code> , <code>sla_deadline_minutes</code> / <code>sla_deadline_time</code> on definitions; <code>sla_at_risk_at</code> / <code>sla_breached_at</code> on <code>JobRun</code>
Webhooks	<code>WebhookEndpoint</code> (<code>webhook_models.py</code>): target job number, hashed token, <code>allow_variables</code> , trigger count
File Mover	<code>MoverRun</code> , <code>MoverOperation</code> , <code>MoverTransfer</code> , <code>MoverLock</code> , <code>MoverAgent</code> , <code>MoverGrant</code> , <code>MoverRelayBudget</code> (<code>file_mover_models.py</code>)
Integrations	Tidal import/export mapping tables, remote-NewWave import mappings, AI monitor tables, simulation tables, version-history tables

Points worth knowing:

- **Inheritance through the tree.** Agent (or agent list), calendar, time window, runtime user, owner workgroup and variables resolve by walking up `parent_group_id` to the nearest ancestor that sets them (`scheduler.resolve_effective_*`). Workgroup visibility for non-admins is resolved the same way (`workgroup_access.py`).
- **A run is a snapshot.** Compiling copies the effective agent, command, working directory, timeout, runtime user and time window onto the `JobRun`. `<Variable>` references are deliberately **not** substituted at that point; they are resolved at dispatch time.
- **Per-run variable values.** A webhook trigger may pass values for the target's own Local (Job) or Group (Job Group) variables. They are stored on each run it creates as `JobRun.variable_overrides` (JSON with the target definition's id) and applied by `resolve_variables` only where a reference resolves to that definition's own variable, so nearest-wins inheritance is unchanged. A context variable plus a SQLAlchemy `init` listener stamps every run created during the trigger (group members, iterations, repeat occurrences) without threading a parameter through every creation path.
- **Attempt history.** Each dispatch of a run writes a `JobRunExecuted` row (attempt number, resolved command, output, exit code, PID), so reruns keep their earlier history.
- **Time.** Timestamps are stored as naive UTC. Jobs carry their own IANA timezone; agents report theirs on each heartbeat for display.

5. Scheduling engine

`scheduler.py` (about 9,000 lines) holds all scheduling logic.

5.1 Compile

`compile_schedule` and `compile_group_schedule` turn definitions into `SCHEDULED` `JobRun` and `PENDING` `GroupRun` rows for every production date from today through `compile_days_ahead`. They evaluate calendars (`calendar_engine.py`), effective dates, offsets, repeat occurrences and iteration. They are idempotent per (definition, date), so the maintenance loop calls them on every wake, keeping the horizon rolling. A day an operator has cleared is recorded in `ClearedScheduleDate` and is not rebuilt automatically.

Taking a Job Group off the schedule is a dated operation, not a side effect of the definition change.

`remove_group_from_schedule` / `remove_job_from_schedule` delete the not-yet-started runs dated on or after a chosen date; `resync_group_schedule` replaces them from a date. Disabling or deleting a group from the console therefore sends `deferScheduleRemoval`: the save changes the definition only and reports

`scheduleRemovalPending`, the UI asks for the date and calls `POST /api/job-groups/{job_no}/remove-from-schedule?fromDate=`, and the descendants follow the same date (`disable-descendants` / `delete-descendants` take `fromDate` or `keepSchedule`). Callers that do not send the flag keep the immediate behaviour. A soft-deleted group that still has future runs cannot be purged (HTTP 409, and `scheduledRuns` on the Recycle Bin rows); `GET /api/job-definitions/deleted-structure` gives Job Activity the names and parents of deleted definitions so their earlier-day runs still nest under them. Restoring a group does not recompile anything: the restore response carries `scheduleGroup` (the group, or its root when nested) and the UI offers the date to `resync-schedule` from.

5.2 Dispatch tick

`run_dispatch_tick` selects `SCHEDULED` and `WAITING_ON_DEPENDENCIES` runs whose production date and scheduled time have arrived, then for each run decides:

1. **Iteration gates** — a sequential iteration waits for its predecessor value.
2. **Time window** — a run whose window has closed becomes `TIMED_OUT` (`READY` runs that were never claimed are checked too).
3. **Dependencies** — `check_dependencies` evaluates `JOB_SUCCESS`, `FILE_EXISTS` and `VARIABLE_VALUE` conditions. Definitions, dependencies and prerequisite statuses for the whole batch are bulk-loaded in a handful of queries rather than per run, which is what keeps large backlogs fast.
4. **If-already-running gate** — skip or wait, per job setting.
5. **Capacity** — `_CapacityBudget` enforces the master's system-wide concurrency cap. When the cap is set it takes a row lock on the master `Agent` row (`UPDLOCK`, `HOLDLOCK` on SQL Server) so concurrent ticks and manual inserts across threads or instances cannot both see room and overshoot. It counts `LAUNCHED` / `RUNNING` plus recently promoted `READY`.
6. **Resource pools** — `_ResourcePoolBudget` (`resource_governance.py`) does the same for each named pool the job needs (§5.6).
7. Result: `READY`, `WAITING_ON_DEPENDENCIES` (with a reason), `AWAITING_APPROVAL` for an approval gate, or `SKIPPED`.

Candidates are ordered by `Job.priority` (higher first) before capacity and pools are consumed, so when several runs compete for the same room the higher-priority one is admitted.

The reason text on a waiting run is structured (`Waiting on: a; b, Waiting for resource pool ...`, `Waiting for scheduled start ...`, ...). Job Activity turns it into specific labels such as *Waiting on Resources* or *Waiting for Time Window* and lists each blocker on its own line.

A **Terminator** job never reaches an agent: once its dependencies are met it finalizes in place and sends a terminate request to its parent group, which cascades down to the group's children.

After the loop, `resolve_pending_agent_list_runs` assigns a concrete agent to runs that were assigned an **agent list** (see §6.6) and `annotate_ready_run_reasons` records why a `READY` run is still waiting.

5.3 Run status machine

Terminal statuses are `COMPLETED_NORMALLY`, `COMPLETED_ABNORMALLY`, `TIMED_OUT`, `ORPHANED`, `ERROR`, `CANCELLED`, `SKIPPED`, `CARRIED` and `TERMINATED`. Every transition goes through `record_run_status_change`, which writes the audit trail and fires status-change events.

5.4 Safety nets that run every tick

- `enforce_max_runtime` raises the max-runtime event for `LAUNCHED / RUNNING` runs past `timeout_seconds` that no agent has reported on. Normally the agent enforces the timeout itself and reports `timed_out` first.
- `reap_stalled_runs` marks a `LAUNCHED / RUNNING` run `ORPHANED` when its agent is offline (no heartbeat within `agent_heartbeat_timeout_seconds`) or no longer exists.
- Deleting a connection is refused while it has `LAUNCHED / RUNNING` runs (`delete_agent`), so runs are not stranded by the delete itself.

5.5 Events and actions

Status changes, file observations, variable changes, timeouts and system conditions fire **events** (`fire_job_event`, `fire_group_event`, `_fire_variable_event`). An event runs an ordered list of **actions** (email, SMS, run a command, set a variable, rerun, job-control actions, and so on) with recursion-depth protection. Email and SMS are sent on background threads so they cannot block a dispatch tick. When AI is enabled and configured, a failure alert gets a short AI diagnosis appended (`scheduler._ai_diagnosis_suffix`, best effort, cached on the run so Job Activity's AI Review opens instantly).

5.6 Resource governance

A `ResourcePool` has a `max_concurrent` number of units. Jobs and job groups declare `JobResourceRequirements` (pool, units); for each pool the nearest definition in the tree that mentions it wins, so a job can override its group. At dispatch, `_ResourcePoolBudget` locks the pool rows, counts units held by `LAUNCHED / RUNNING` runs plus runs promoted to `READY` within a 30 s grace window, and reserves in memory for the rest of the batch. A run that cannot get its units waits with a `Waiting for resource pool` reason. When no pool has any requirement the budget costs one cheap query per tick.

5.7 Control nodes: approval gate and branch

Both are `utility_type` values that never reach an agent.

- **Approval gate** (`approval_gate`): once its dependencies are met the run becomes `AWAITING_APPROVAL`. `POST /api/job-runs/{id}/approve` completes it normally (releasing dependents); `/reject` completes it

abnormally. Only the gate's listed approver usernames or workgroups, or administrators, may decide. It can be cancelled like a waiting run.

- **Branch** (`branch`): runs once its upstream job has finished, whatever the outcome. Its `BranchRule`s (exit-code comparisons, plus an optional default) are evaluated in order against the upstream exit code; the first match picks the target, and every other target is set `SKIPPED`.

5.8 SLA deadlines

A job or group may set `sla_deadline_minutes` (finish within N minutes of starting) and/or `sla_deadline_time` (finish by a time of day in the schedule timezone); both inherit down the tree. `check_sla_breaches` raises one alert when 80 % of the window has elapsed and one when it is missed, deduplicated by `JobRun.sla_at_risk_at` / `sla_breached_at`.

5.9 Critical path

`GET /api/job-definitions/{root}/critical-path` (`critical_path.py`) builds the dependency DAG under a group, weights each job by its average recent run duration, and returns the longest weighted path (back-edges are ignored, so a cycle cannot loop). Workflow Designer outlines that path when *Critical Path* is on.

6. Agents

An **agent** is a named execution endpoint. The server has two kinds, and both consume work through the same claim function.

	External agent (<code>newwave_agent</code>)	Built-in Master agent
Where it runs	A remote Windows, Linux or macOS host	Inside the server process
How it gets work	Polls <code>GET /api/v1/internal/agents/{name}/work</code>	<code>local_agent.local_execution_tick()</code> calls <code>claim_next_ready_run</code> directly
Executes with	<code>agent/executor.py</code>	<code>local_executor.py</code> , a port of the same executor kept in step by hand
Row in <code>agents</code>	Auto-created on first contact	<code>is_master = true</code> , name <code>Master</code> (this server), created at startup
Concurrency cap	<code>NW_AGENT_MAX_WORKERS</code> (20) and the server-reported capacity	<code>NW_LOCAL_EXECUTION_MAX_CONCURRENCY</code> (8)
Can be disabled	Stop the connection	System Settings → Built-in local execution

A real agent may also be pointed at the Master's name. The atomic claim (see §7.3) means both paths can coexist without double-running a run.

6.1 Agent components

The agent is a standalone Python package (`agent/` , with shared schemas in `common/schemas.py`):

Module	Responsibility
<code>main.py</code>	Entry point. Intercepts <code>ftp</code> as the first argument <i>before</i> importing anything else, so a one-shot <code>nw-agent ftp ...</code> call inside a job does not start the agent.
<code>winservice.py</code>	Windows service <code>NewWaveAgent</code> (pywin32)
<code>service.py</code>	<code>AgentService</code> : builds one <code>SchedulerConnection</code> per configured server, owns the shared work queue, the worker tasks and the pending-results retry loop
<code>scheduler_connection.py</code>	Everything about one scheduler: HTTP calls, failover state, and the heartbeat, poll, cancel-poll and peer-check loops
<code>executor.py</code>	Runs one command and returns an <code>ExecutionResult</code>
<code>config.py</code>	<code>NW_AGENT_*</code> settings (<code>pydantic-settings</code>)
<code>pending_results.py</code>	On-disk queue of results the server did not acknowledge
<code>peer_cache.py</code>	On-disk cache of each server's HA peer URL
<code>webserver.py</code>	Read-only local troubleshooting UI
<code>ftpserver.py</code> , <code>sftpserver.py</code> , <code>ftp_certs.py</code>	Optional inbound file-transfer servers
<code>ftp_cli.py</code>	<code>nw-agent ftp</code> client used from job commands
<code>file_mover/</code>	Built-in File Mover worker: <code>worker.py</code> (polling, leases, receipts, concurrency, state pruning), <code>ops.py</code> (capture, stage, verify, backup, publish, service, command, rollback, ...), <code>buffer.py</code> (bounded command output). See §10.1

6.2 Concurrency model

One asyncio event loop. Per configured scheduler there are four independent loops; across all schedulers there is one bounded queue and a fixed pool of worker tasks.

- Only the **poller** talks to `/work` , so an idle agent makes one request per poll interval rather than one per worker. Workers only `await queue.get()` and cost nothing while idle.
- The poller stops claiming when `in_flight >= capacity` , where `capacity` is the value the server returned on the last heartbeat (`max_workers` until the first heartbeat succeeds). A claimed run counts as in flight from the moment the server returns it, because the server has already set it `LAUNCHED` .
- Two limits therefore apply: a soft per-scheduler cap (server-reported capacity) and a hard total across all schedulers (`max_workers` , enforced by the bounded queue and fixed worker count).

6.3 Multiple schedulers and failover

An agent can serve several independent NewWave servers at once

(`NW_AGENT_SERVER_URLS=http://a:8010,http://b:8010`). Each URL gets its own `SchedulerConnection` with its own queue position, in-flight count, cancel events, PID inventory and failover state. Nothing is special about the first URL beyond a one-time migration of the legacy peer-cache file.

Failover to an HA peer (see §9) is per scheduler:

- The agent learns the peer URL by asking the server (`/scheduler-peer` , every `NW_AGENT_PEER_CHECK_INTERVAL = 30 min`) and persists it. There is no statically configured standby address.
- After **3 consecutive connect failures** — counted across all four loops of that connection — it switches `active_url` to the peer.
- While on the peer, each heartbeat also probes the primary and switches back when it answers.
- Connection-level failures back off exponentially from 2 s up to 60 s with $\pm 50\%$ jitter, so a fleet does not reconnect in lockstep after a server restart. Application errors (HTTP 4xx/5xx) do not back off or fail over.

6.4 Executing a command

`executor.execute(request, cancel_event, on_progress, on_started) :`

1. Builds the environment: the agent's own environment plus `request.environment` (used for plugin secrets).
2. **POSIX:** runs the command line through `/bin/sh -c` in its own session (`start_new_session=True`), so shell syntax (`&` , `;` , pipes, redirects) works and the whole process group can be killed. If `run_as_user` is set, it uses `Popen(user=...)` , which requires the agent to have privilege to switch users; otherwise the run fails to start rather than silently running as the agent's account.
3. **Windows:** runs the program directly with no shell by default. A shell can be chosen globally (`NW_AGENT_WINDOWS_SHELL : none , cmd , powershell , pwsh` or an absolute path) or per job. With a `run_as_user` and a password it uses a separate path (`LogonUser` and `CreateProcessAsUser`); with a user but no password it logs a warning and runs as the agent's own account.
4. A built-in `sleep <n>` is executed without spawning a process (used by the demo and by tests).
5. Output is read incrementally from both pipes. Every 10 seconds (`PROGRESS_INTERVAL_SECONDS`) the accumulated output is pushed to the server as a live tail.
6. It waits for whichever comes first: process exit, cancellation, or `timeout_seconds` .
7. On cancel or timeout the whole process tree is killed (`taskkill /F /T` on Windows, `killpg(SIGKILL)` on POSIX) and the pipes are drained for up to 5 s.
8. Returns `ExecutionResult` with status `success` , `failed` , `timed_out` or `cancelled` , the exit code, output, PID and timestamps.

The agent reports only what happened. **The server decides success or failure** (§7.5).

6.5 Reliability

Failure	Behaviour
Server unreachable when a result is ready	Result is written to <code>pending_results/</code> and retried every 60 s for up to 24 h, surviving agent restarts. Each file records which scheduler it belongs to.
Server rejects a result with 404 or 409	The run is gone or already terminal; the result is dropped, not retried.
Agent service stops mid-run	<code>execute()</code> kills the process; the worker reports <code>FAILED</code> ("Agent service stopped...") before exiting.
Started-confirmation or progress call fails	Ignored. The next heartbeat's PID inventory repairs the server's view.
Agent process crashes	The server sees no heartbeats and, once the timeout elapses, <code>reap_stalled_runs</code> marks its runs <code>ORPHANED</code> .

6.6 Agent lists

A job can be assigned an **agent list** instead of one agent. Because the right choice depends on which members are online *now*, a list-assigned run is compiled with `agent_name = NULL` and `agent_list_name` set, and `agentlist_scheduling.resolve_pending_agent_list_runs` picks a concrete agent each tick once the run is `READY`. Strategies include ordered, random and balanced (least active runs relative to `Agent.capacity`) selection, plus broadcast to several agents. Only ONLINE members are candidates. After resolution the run is indistinguishable from a plain-agent run, so the claim path is unchanged.

6.7 Optional agent services

Both are **off by default** and share `ftp_server_root_dir`, username and password.

- **Local web UI** (`webserver.py`): read-only, on `127.0.0.1:8765` by default, showing the last 200 runs and their output, and the agent log.
- **FTP/FTPS server** (`pyftplib`, port 2121, explicit TLS with an auto-generated self-signed certificate if none is supplied) and **SFTP server** (`paramiko`, port 2222). These are remote-writable file surfaces with plaintext-configured credentials, hence the explicit opt-in.
- `nw-agent ftp` **CLI**: an FTP/FTPS/SFTP client for use inside job commands. FTP jobs on the server are compiled into these commands.

6.8 Agent configuration

All settings are environment variables with the prefix `NW_AGENT_` (or a `.env` file).

Setting	Default	Meaning
<code>NAME</code>	host name	Identity; the agent is registered under this name
<code>TOKEN</code>	empty	Bearer token sent on every call

Setting	Default	Meaning
<code>SERVER_URLS</code>	—	Comma-separated scheduler URLs (preferred)
<code>SERVER_URL</code> , <code>ADDITIONAL_SERVER_URLS</code>	<code>http://localhost:8000</code>	Legacy equivalents
<code>POLL_INTERVAL</code>	5 s	Idle work-poll interval
<code>HEARTBEAT_INTERVAL</code>	30 s	Heartbeat interval
<code>MAX_WORKERS</code>	20	Hard concurrency ceiling
<code>PEER_CHECK_INTERVAL</code>	1800 s	HA peer lookup; 0 disables
<code>TAGS</code>	<code>os=<platform></code>	<code>key=value</code> pairs reported on heartbeat
<code>TIMEZONE</code>	auto-detected	IANA zone reported on heartbeat
<code>WINDOWS_SHELL</code>	<code>none</code>	Default Windows shell
<code>WEB_ENABLED</code> , <code>WEB_HOST</code> , <code>WEB_PORT</code>	<code>true</code> , <code>127.0.0.1</code> , 8765	Local UI
<code>FTP_SERVER_*</code> , <code>SFTP_SERVER_*</code>	<code>off</code>	File-transfer servers
<code>PENDING_RESULTS_DIR</code> , <code>LOG_DIR</code>	under <code>ProgramData\NewWaveAgent</code> (Windows) or <code>/var/lib/newwave-agent</code>	State and logs
<code>FILE_MOVER_ENABLED</code>	<code>true</code>	File Mover worker; it only runs once <code>TOKEN</code> is set
<code>FILE_MOVER_STATE_DIR</code>	<code>file-mover</code> under the same base	Operation receipts and recovery journals; keep across upgrades
<code>FILE_MOVER_CONCURRENCY</code>	2	File Mover operations worked at once (from different runs)
<code>FILE_MOVER_STATE_RETENTION_DAYS</code>	14	Age at which receipts and finished journals are pruned

6.9 Packaging

- **Windows:** built with PyInstaller (`deploy/build-agent-exe.ps1`) and installed as the `NewWaveAgent` Windows service by `deploy/install-agent-windows.ps1` .
- **Linux/Unix:** `deploy/install-agent-linux.sh` creates a Python virtualenv and installs a `newwave-agent` systemd unit running as an `nwagent` user. Running jobs as other users needs `CAP_SETUID` (typically root), and killing such jobs needs `CAP_KILL` .
- `agent/_version.py` is stamped at build time; the version and start time are sent on each heartbeat and shown on the Connections screen.

7. Agent protocol

The agent-facing endpoints live in `agent_protocol.py` under the prefix `/api/v1`. The prefix is hard-coded in deployed agents, so it cannot change without updating them. All calls are JSON over HTTP(S), initiated by the agent.

Method and path	Sent by agent	Purpose
POST <code>/agents/{name}/heartbeat</code>	Every 30 s	Liveness, metadata, PID inventory; returns <code>capacity</code>
GET <code>/agents/{name}/scheduler-peer</code>	Every 30 min	Learn the HA peer URL
GET <code>/internal/agents/{name}/work</code>	Every 5 s when idle	Claim the next <code>READY</code> run; returns an <code>ExecutionRequest</code> or <code>null</code>
POST <code>/runs/{id}/started</code>	Once, after the process starts	Body <code>{pid}</code> ; <code>LAUNCHED</code> → <code>RUNNING</code>
POST <code>/runs/{id}/progress</code>	Every 10 s while running	Live stdout/stderr tail
GET <code>/internal/agents/{name}/cancellations</code>	Every 5 s while runs are in flight	Run IDs to kill
POST <code>/runs/{id}/result</code>	Once, at the end	Final <code>ExecutionResult</code>
POST <code>/file-mover/agents/{name}/poll</code>	Every 3 s while a File Mover slot is free	Claim a File Mover operation (token required)
POST <code>/file-mover/agents/{name}/operations/{id}/heartbeat</code>	Every 15 s per operation	Renew the 60 s operation lease; learn about cancellation
POST <code>/file-mover/agents/{name}/operations/{id}/result</code>	Once per operation	Operation outcome (<code>SUCCESS</code> , <code>FAILED</code> , <code>UNCERTAIN</code>)
<code>/file-mover/agents/{name}/operations/{id}/transfers/...</code>	During capture and stage	Relay upload and download in 8 MiB chunks (create, chunk, complete, status)

The webhook trigger `POST /api/v1/webhooks/{id}/trigger` shares the `/api/v1` prefix but is called by external systems, not agents (§10.2).

7.1 Registration and heartbeat

There is no enrollment step. The first call under a new name creates the `Agent` row (`_get_or_create_agent`), with capacity 1 and a generated token. New names are subject to the licence's `maxAgents`; agents that already exist are never cut off retroactively. An administrator who wants token authentication pre-creates the agent (`POST /api/agents`), which returns the token once, or issues a new token for an existing agent with **Generate Token** on its Connections form (`POST /api/agents/{id}/rotate-token`; the old token stops working immediately).

The heartbeat body (`AgentHeartbeat`) carries host name, OS, tags, agent version, start time, IANA timezone, `active_run_pids`, and `capabilities` (for example `{"fileMover": 1}`). The server:

- marks the agent ONLINE and stamps `last_heartbeat`;
- overwrites hostname, OS, tags and version; overwrites the stored timezone when the agent supplies one;
- records File Mover support in `file_mover_agents`, including whether this heartbeat carried the agent's valid token (`token_ok`). The Connections screen derives *Ready*, *Needs token*, *Offline* or *Not supported* from it, and the File Mover engine refuses to start a workflow on an agent that is not ready;
- **reconciles processes** (`_reconcile_agent_processes`), treating the agent as authoritative:
 - a `RUNNING` run with no PID in the map becomes `ORPHANED`;
 - a `LAUNCHED` run whose PID is reported becomes `RUNNING` (repairing a lost `/started` call).

Older agents that omit `active_run_pids` (`null`) skip this. An empty map means "no active jobs".

The response returns the server-side `capacity`, which sets the agent's per-scheduler poll throttle. Status shown in the UI is derived: OFFLINE when `last_heartbeat` is older than `agent_heartbeat_timeout_seconds`, not the stored column. A work poll also counts as a heartbeat (writes are throttled to one per 3 s).

7.2 Message shapes

```
// GET .../work → ExecutionRequest (or null)
{
  "run_id": 123,
  "job_id": "uuid-or-run-id",
  "command": "backup.sh --full",           // command + parameters, variables resolved
  "arguments": [],                         // unused; the command is one string
  "working_dir": "/data",                  // variables resolved
  "environment": {},                       // plugin secrets, agent-mode plugins only
  "timeout_seconds": 3600,
  "run_as_user": "svc_batch",               // frozen onto the run at compile time
  "run_as_password": "...",                // looked up fresh at dispatch time
  "is_powershell": false,
  "shell": null                           // "powershell" for PowerShell jobs
}

// POST .../result ← ExecutionResult
{
  "run_id": 123,
  "status": "success | failed | timed_out | cancelled",
  "exit_code": 0, "stdout": "...", "stderr": "...", "pid": 4242,
  "started_at": "...", "finished_at": "...", "error_message": null
}
```

7.3 Work claim

`claim_next_ready_run(session, agent)` is shared by the HTTP endpoint and the Master's local loop. It:

1. selects the oldest `READY` run whose `agent_name` equals this agent;
2. **claims it atomically** with a conditional `UPDATE job_runs SET status='LAUNCHED' ... WHERE id=? AND status='READY'` and proceeds only if `rowcount == 1`. Otherwise another poller won and it tries the next run. This closes the select-then-write race that once dispatched the same run twice;
3. resolves `<Variable>` references in command, parameters and working directory against the current variable values. If any is undefined the run becomes `ERROR` and is skipped, so one bad run does not starve others;
4. for FTP jobs, fills in the password from a Runtime User; for plugin jobs in agent mode, asks the adapter to build the real command and a secret-bearing environment (§10);
5. writes a `JobRunExecuted` attempt row;
6. looks up the runtime user's password fresh, keyed by the agent's OS (`windows` or `unix`), so a password rotation applies to every run not yet dispatched and the password is never stored on the run;
7. returns the `ExecutionRequest`.

Callers apply their own gates first. `get_work` returns `null` (HTTP 200) for a **stopped** agent, and when the licence does not permit scheduling; the Master's loop applies the same gates plus the HA lease.

7.4 Cancellation

Cancelling a `LAUNCHED / RUNNING` run (`scheduler.cancel_run`) sets `cancel_requested` and a status reason but does not change the status. The agent's cancel-poll loop, active only while it has runs in flight, fetches the IDs to stop, sets the matching `asyncio.Event`, and `execute()` kills the process tree and reports `cancelled`. The server keeps the requester's name in the final reason. For Master-executed runs the same call reaches `local_agent.request_cancel` directly, with no polling.

7.5 Result and success evaluation

`receive_result` → `finalize_run_result`:

- Returns HTTP **409** if the run is not `LAUNCHED / RUNNING`, and **404** if it does not exist.
- For `success` and `failed`, the server **re-derives** the outcome from the job's configured Success Criteria (`evaluate_job_success`: exit-code ranges and output-text checks). The agent's own verdict is not trusted. `timed_out` and `cancelled` are taken as reported.
- Closes the open `JobRunExecuted` attempt, stores exit code, output and PID, and records the status change (which fires events and actions).
- Handles agent-reported timeouts, file-check and file-event results, plugin output extraction into variables, and "run same occurrence after execution" reruns.
- Sequential list iterations trigger an immediate dispatch tick so the next value starts without waiting for the loop.

`finalize_run_result` re-checks the run's status itself because the Master's local loop calls it too; a run the reaper already orphaned is never overwritten by a late result.

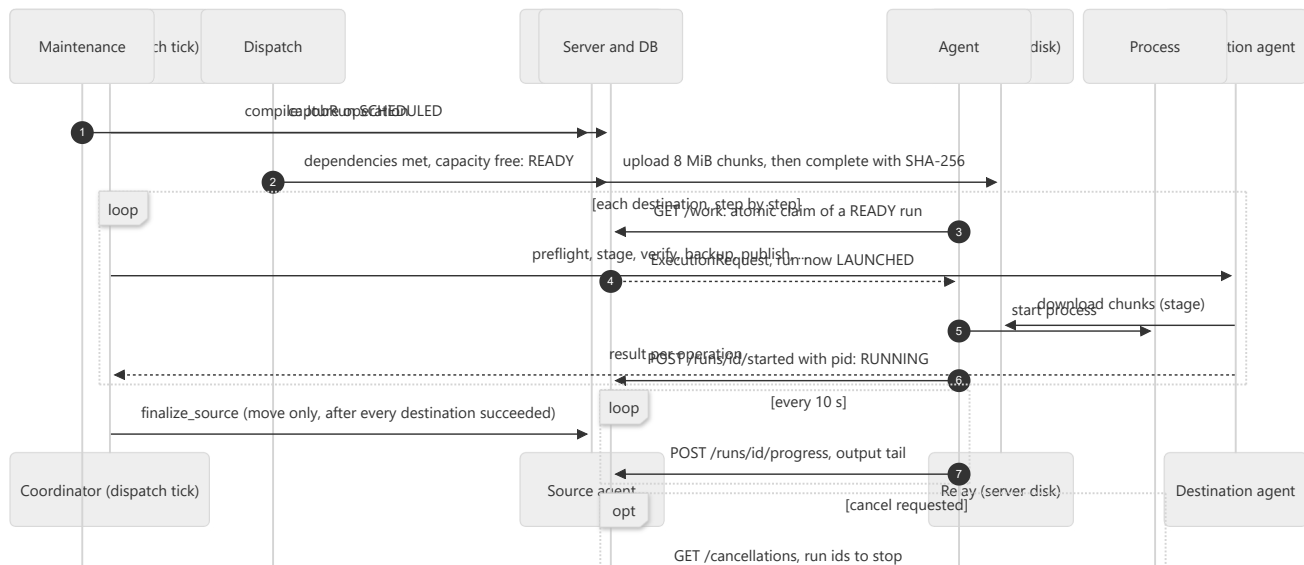
7.6 Synthetic runs: file checks and file events

Some server features need a fact from an agent's host. They are implemented as **synthetic** `JobRuns` dispatched through this same pipeline:

- **File-exists dependencies** dispatch a small OS-specific command (PowerShell on Windows; `stat` on Linux and macOS; a portable fallback otherwise) that prints `EXISTS <size> <mtime>` or `MISSING`. The result updates `FileCheckCache`.
- **File events** dispatch directory scans the same way and turn observations into firings.

Because they reuse the ordinary claim/execute/report path, agents need no special support, and they respect the agent's stopped, capacity and licensing gates like any other run.

8. Run lifecycle end to end



9. High availability

Optional active/standby scheduling, enabled in System Settings. It requires two or more server processes pointing at the **same database**.

- `SchedulerLease` is a singleton row. `try_acquire_or_renew_lease` renews with `UPDATE ... WHERE holder = me`; if that affects no rows, it tries to take the lease with `UPDATE ... WHERE holder IS NULL OR heartbeat < now - timeout`. Both are single conditional statements decided by `rowcount`, so two instances cannot both win.
- The **dispatch**, **maintenance** and **local-execution** loops all check the lease each tick. A standby instance does nothing (`STANDBY`) but keeps ticking, so it takes over once the holder's heartbeat is older than `ha_lease_timeout_seconds` (default 30 s).
- The Master agent executes locally only on the lease holder, otherwise both instances would run every Master-assigned job.

- **Agents** are told the peer's address via `peer_base_url` (§6.3) and fail over on their own. Both instances serve the agent API from the shared database, so an agent that fails over sees the same runs.
- Each instance has a fresh `INSTANCE_ID` per process start; a restarted process is a new contender, never assumed to still hold the lease.

10. Plugins, job types and integrations

Job types are variations on one thing: producing a command for an agent (or, for in-process plugins, running code in the server).

utility_type	How it runs
<code>os_command</code>	The command and parameters run as written
<code>ftp</code>	Compiled (<code>utility_builders.py</code>) into an <code>nw-agent ftp ...</code> command; the password may come from a Runtime User at dispatch
<code>terminator</code>	Never dispatched; finalizes in place (§5.2)
<code>approval_gate</code> , <code>branch</code>	Never dispatched; control nodes (§5.7)
<code>plugin</code>	A named adapter, see below

Plugin adapters (`plugins/`) integrate third-party systems without touching the scheduler, agent protocol or frontend. An adapter subclasses `PluginAdapter`, declares its job fields and connection fields as JSON Schema (the UI renders them generically), and is discovered at startup from either an installed package exposing a `newwave.plugins` entry point or `plugins/builtin/` (currently `echo`, `restapi`, `sqlserver`, `jdbc`, `scp`, `orchestration`, `file_mover`). A bad adapter is logged and skipped, never fatal.

A plugin job chooses an **execution mode**:

- `in_process` — the adapter runs in the server (`plugin_runtime.py`, executed on a worker thread with a timeout and a cooperative cancel event), driven by the Master's local loop. Adapters get no database session and cannot touch NewWave's schema.
- `agent` — at dispatch the adapter's `build_agent_command` and `agent_environment` produce a real command and environment for the remote agent. Connection secrets are decrypted (`crypto.py`) and delivered through the environment, so they never enter the command string or the stored attempt row. The dispatcher passes the claiming agent's name and OS in `PluginContext` (`agent_os`), so an adapter can quote for that machine, and an adapter that needs a script on a Windows agent returns `True` from `agent_uses_powershell` (the request then runs through PowerShell for that dispatch only). `scp` is agent-only: key login is a plain `scp` command line; password login is a small askpass script with the password in `NW_SSH_PASSWORD`. Adapters may implement `preview_command`, served by `POST /api/plugins/{type}/preview` for the job form's live command preview (no secrets returned). Operator guide: `docs/SCP.md`.

- `orchestrated` — the server coordinates a multi-step workflow across several agents over a separate protocol. Used by File Mover only.

Named **plugin connections** hold non-secret config in plaintext and secrets encrypted, and can source credentials from a Runtime User.

10.1 File Mover

Copies or moves one captured file (up to 10 GiB) from a source agent to one or more destination agents, with optional deployment steps. Server side: `file_mover_spec.py` (workflow schema and validation), `file_mover_engine.py` (coordinator), `file_mover_api.py` (UI API and agent API), `file_mover_access.py` (workgroup and path grants). Agent side: `agent/file_mover/` in `newwave_agent`. Operator guide: `docs/FILE_MOVER.md`.

- **Coordination.** A `MoverRun` holds the immutable workflow snapshot and its progress. The coordinator runs inside the dispatch tick, advances each active run under a compare-and-swap on `MoverRun.revision` (safe across HA instances), and creates **one** `MoverOperation` **at a time per run**. Path and service locks (`MoverLock`) keep conflicting runs apart.
- **Operations and leases.** An agent claims an operation by polling; the claim is a conditional update that gives it a 60 s lease, renewed by a heartbeat every 15 s. A leased operation doesn't block the agent's other runs: the poll skips it, and the agent works up to `FILE_MOVER_CONCURRENCY` operations at once. The agent tolerates heartbeat failures for 40 s before treating authority as lost, and stops at once if the server rejects the heartbeat.
- **Receipts and journals.** Before acting, the agent writes a receipt; file mutations go through a write-ahead journal (stage, backup, publish intent, renames, created directories, stopped services, commands and their compensations). A lost acknowledgement is answered from the receipt; a restart during a safe operation resumes; a restart during a non-idempotent command is reported `UNCERTAIN` rather than replayed.
- **Relay.** Artifacts pass through the server's relay directory in 8 MiB chunks, each verified by SHA-256. The agent hashes the whole file while uploading and sends the digest with `complete`, where the server re-verifies the stored chunks. Space is reserved with an atomic quota update (`MoverRelayBudget`). Chunk storage runs off the event loop. Successful artifacts are removed on completion; failed or cancelled ones after 24 h. Master Status shows the relay path, reserved space and free disk.
- **Recovery.** On failure or cancellation, rollback operations restore the active destination from its backup, run command compensations, remove created directories, and return services to their previous state. Five committed backups are kept per destination file.

- **Credentials.** A poll returns only the runtime-identity passwords that operation can use: a command's own identity and its compensation's, or, for a rollback, the compensation identities of that run's commands on that agent.
- **Readiness.** Agents advertise the protocol in their heartbeat; File Mover always requires the agent token (§7.1).

10.2 Inbound webhooks

`WebhookEndpoint` rows (Administration → Webhooks) each trigger exactly one job or group. `POST /api/v1/webhooks/{id}/trigger` with `Authorization: Bearer nwh_...`:

- Authentication (`webhook_trigger.py`) compares the SHA-256 of the token in constant time; unknown id, disabled endpoint and wrong token all return the same bare 404. Failed attempts per client IP and all calls per endpoint are rate limited (429 with `Retry-After`).
- The body is read with a 16 KiB cap before anything is parsed. If the endpoint allows variables, `{"variables": {...}}` may set values for the target's own variables (at most 50, each at most 4,000 characters, no `<`, `>` or control characters, since substituted values are re-scanned for references). Values apply to that trigger's runs only (§4).
- The trigger runs `run_job_now` / `run_group_now`, updates the endpoint's counters, and writes an audit entry including the variables passed.

10.3 AI-assisted features

Optional, behind **Enable AI** (System Settings) and an API key on AI Monitor. `ai_builder.py` turns a plain-English description into a proposed job (`POST /api/ai/propose-job`); every agent, calendar or dependency the model proposes is validated against what exists before the proposal is returned, and nothing is saved until the user saves the form. `ai_review.py` produces run reviews and the diagnosis appended to failure alerts (§5.5).

11. Security

Concern	Mechanism
Interactive login	Local accounts (PBKDF2-HMAC-SHA256, 600,000 iterations, per-user salt) or LDAP; signed session cookie (<code>SECRET_KEY</code>), 15-minute default lifetime
Programmatic access	API tokens with prefix <code>nwt_</code> ; only a SHA-256 hash is stored and the token is shown once
Webhooks	Per-endpoint <code>nwh_</code> tokens (hash only, shown once) that can trigger one job or group; uniform 404 on any auth failure, per-IP and per-endpoint rate limits, 16 KiB body cap, audit logged (§10.2)
Authorization	Role checks in <code>auth_gate_middleware</code> and <code>require_admin</code> ; workgroup ownership for non-admins (<code>workgroup_access.py</code>)
Agent authentication	<code>Authorization: Bearer <agent token></code> , checked against <code>Agent.token</code> with a constant-time compare. Optional for the File Mover agent API while <code>AGENT_AUTH_REQUIRED</code> is off (a token that is sent must be valid); required once it is on

Concern	Mechanism
File Mover	Workgroup-scoped path grants checked at authoring and again at dispatch; paths re-checked on the agent (symlinks and junctions refused); credentials sent only to the operation that uses them (§10.1)
Run as another user (Linux standalone)	The small setuid-capable helper <code>idruns-runas</code> (<code>deploy/runas_helper.c</code>) carries <code>cap_setuid, cap_setgid</code> ; the server executable itself stays unprivileged. The helper refuses uid 0 and is group-restricted
Licence	Ed25519-signed blob; enforcement pauses scheduling only, never login or viewing
Secrets at rest	Plugin connection secrets encrypted (<code>crypto.py</code>)
Audit	Field-level audit log for entity changes and status transitions

12. Frontend

A single-page app in `src/` , built by Vite into `dist/` and served by the backend.

```
src/
  main.tsx          providers: Theme → QueryClient → Router → Auth
  App.tsx           routes; route guards for admin-only and demo
  app/              shell: three layouts (Command Center, Classic, Sidebar),
                    nav.ts (menu and feature-flag gating), AuthContext
  components/       DataGrid, Modal, Tabs, pickers, version history
  screens/          one folder per area (operations, jobs, connections, ...)
  data/repository.ts typed API client; every server call goes through it
  data/gridRegistry.ts column definitions per grid
  theme/            ThemeContext and tokens.css (nine colour schemes)
  types/            shared TypeScript types
```

- **Server state** is fetched with TanStack Query (with per-screen polling where freshness matters, e.g. Connections every 10 s). The client holds no authoritative state beyond UI preferences.
- **Layout** is chosen by the administrator (System Settings) and applies to everyone; the **colour scheme** is per browser, except in demo mode where it is fixed. Menu entries can be hidden by role and by feature flags (`enableTidalImport` , `enableScheduleSimulator` , ...).
- `DataGrid` is the shared table: sorting, filtering, grouping, resizable and reorderable columns, tree rows, row selection, and a toolbar. It persists sort, filters, layout and selection toggles per grid title in `localStorage` .
- **Flow views** (Job Activity Flow, Workflow Designer) use `@xyflow/react` with `dagre` for automatic layout.
- **Definition editors** are modal forms with tabs; the Job form validates ownership and inheritance before save. A definition the user may not change opens read-only; Job Activity's run details link straight to it and return to the run on close.

- **Dependency lines** (`DependencyLinesOverlay.tsx`) draw connectors between grid rows on Jobs and Job Activity. Cross-group dependencies are dashed; inside *Go Into Group*, a dependency on a job outside the scope is drawn as a stub ending in a hoverable circle naming the outside job.
- **Column filters** support value groups (e.g. *All waiting* on Job Activity's status) and per-column value normalization, so display-only variants such as `Completed Normally *` filter as one value.
- **First-run onboarding** (`OnboardingGuide.tsx`) walks a new user through the console; progress is kept per browser.

13. Deployment and operations

Server (Windows). `service/install-service.ps1` builds the frontend and registers `python -m uvicorn app.main:app --host 0.0.0.0 --port 8010` as the NSSM service `IDRuns` (formerly `NewWave`), logging under `logs/`. A frontend-only change needs only `npm run build`; a backend change needs `Restart-Service IDRuns`.

NSSM rotates `service.out.log / service.err.log` on every start and at 10 MB. The service is installed with `AppThrottle 30000` and `AppRestartDelay 10000`, so a failing start is retried every 10 s or so rather than in a tight loop. The maintenance loop sweeps the rotated files on every wake: past `service_log_retention_days` (7), near-empty restart stubs, and anything beyond the newest `SERVICE_LOG_MAX_FILES` (20).

Server (other). `backend/README.md` covers a macOS/Linux install. The database is created and upgraded with `python -m alembic upgrade head`. The server does not migrate on start: apply migrations **before** restarting onto code that needs them, otherwise every query on an affected table fails until they are applied.

Standalone edition. `deploy/build-exe.ps1` (Windows) and `deploy/build-exe.sh` (Linux) build one PyInstaller executable (`idruns`) containing the Python runtime, backend, built frontend and the migration chain. On first launch `deploy/launcher.py` asks for a data directory, runs a setup wizard (optionally creating an administrator), writes `<data-dir>/idruns.env`, migrates a fresh SQLite database and starts the server. The product name and tagline are fixed (`IDRuns / Workload Automation`), never asked. `idruns install-service` registers it as a service. Licensing starts the 30-day grace period on first start. A remote Linux server is updated with `deploy/deploy-idruns-server.sh` (or the `.ps1` from Windows), which copies the binary - taken from `deploy/dist/idruns`, where `build-exe.sh` puts it - and runs `deploy/idruns-server-update.sh` there: back up, swap, restart, and roll back on failure. On Linux, running jobs as another user uses the separate `idruns-runas` helper (see §11; `deploy/setup-runas-helper.sh` installs it).

Documentation PDFs. `docs/*.pdf` (User Guide, Tutorial, both Architecture documents, Product Overview) are generated from their `.md` files by `scripts/docs-pdf.mjs`: Markdown to HTML with `marked`, Mermaid diagrams drawn by `mermaid`, then printed by headless Chrome, Edge or Chromium (set `CHROME_PATH` to pick one). `npm run docs:pdf` rebuilds the PDFs older than their source, `npm run docs:pdf:watch` rebuilds on every save, and `npm run docs:pdf:check` fails when one is stale. `npm run docs:hooks` turns on `.githooks/pre-commit`, which rebuilds and stages the PDF of every committed `.md` so the two never drift. A new document gets a PDF by adding its name to `NAMES` in the script. Each PDF that is built is also copied into the website checkout's `assets/docs/` (`../idruns_website`, or `IDRUNS_WEBSITE_DIR`) under its published name, and `node scripts/docs-pdf.mjs --sync` copies without rebuilding; the download cards are in the website's `index.html`.

Database. SQLite (WAL, foreign keys on, 5 s busy timeout) for development, single-node and standalone use; it now also uses a `QueuePool` (40 + 80 overflow) so concurrent checkouts reuse connections instead of opening one per request. SQL Server uses the same pool sizing, because agent polling, progress and result traffic plus request sessions can burst past SQLAlchemy's default of 15 after a restart.

Agents. Installed per host as described in §6.9. An agent needs only outbound HTTP(S) access to the server (and to a peer, if HA is configured), and a name that is unique across the fleet. File Mover is part of every current agent build; it activates once the agent's token is set (Connections → agent → Generate Token, then `NW_AGENT_TOKEN`).

Logging. The server writes one rotating `backend.log` (uvicorn's own logs are folded into it); the agent writes its own rotating log. Rotated service logs are swept as described above.

Tests. `backend/tests/` holds the pytest suite (50 test modules). The test run points `LOG_DIR`, `LOG_FILE` and `SERVICE_LOG_DIR` at a temporary folder (`conftest.py`), so it never writes to or purges the live service's logs. `test_file_mover.py` drives the real agent operations from a sibling `newwave_agent` checkout (`NW_AGENT_REPO` overrides; it skips if absent); the agent's own File Mover tests are `newwave_agent/test_file_mover_worker.py`. Some scheduler tests are anchored to the wall clock and are unreliable near 23:00–00:00 UTC.

14. Known limitations

These follow from the code as it stands and are worth deciding on before a wider rollout.

1. **Agent authentication is off by default.** With `AGENT_AUTH_REQUIRED=false` a missing or wrong token is logged but accepted, and a caller can register a new agent name just by contacting the server. Enforcement should be enabled once every agent has its token.
2. **Agent tokens are stored in plaintext** (`Agent.token`), unlike API tokens, which are hashed.
3. **Credentials cross the agent link in clear text.** `run_as_password` is sent in the `ExecutionRequest`, and FTP-server credentials are configured in plaintext on the agent. Run the agent link over HTTPS.
4. **Two executors kept in step by hand.** `local_executor.py` is a copy of `agent/executor.py`; a change to one must be copied to the other.
5. **SQLite is single-process.** HA and multi-process operation require SQL Server; the capacity-lock and lease logic rely on it.
6. **Very large single modules.** `scheduler.py` (~9,300 lines) and `main.py` (~11,300 lines) hold most logic; navigating them relies on their docstrings.
7. **File Mover relays through the server.** Every byte goes source agent → server disk → destination agent, so the relay's disk and the server's bandwidth bound throughput. Direct agent-to-agent transfer is not implemented.
8. **No automatic migration on start.** Deploying code ahead of its migration makes the affected queries fail until `alembic upgrade head` is run; with NSSM this shows up as a (now throttled) restart loop.