Workgraph

A Manual

Task coordination for humans and AI agents

Contents

1 Glossary

The following terms have precise meanings throughout this manual. They are defined here for reference and used consistently in every section.

Term	Definition
task	The fundamental unit of work. Has an ID, title, status, and may have dependencies, skills, inputs, deliverables, and other metadata. Tasks are nodes in the graph.
status	The lifecycle state of a task. One of: *open* (available for work), *in-progress* (claimed by an agent), *done* (completed successfully), *failed* (attempted and failed; retryable), *abandoned* (permanently dropped), or *blocked* (explicit, rarely used). The three *terminal* statuses are done, failed, and abandoned—a terminal task no longer blocks its dependents.
dependency	A directed edge between tasks expressed via `blocked_by`. Task B depends on task A means B cannot be ready until A reaches a terminal status.
blocked_by	The authoritative dependency list on a task. A task is *blocked* (in the derived sense) when any entry in its `blocked_by` list is non-terminal.
blocks	The inverse of `blocked_by`, maintained for bidirectional traversal. Not checked by the scheduler—purely a convenience index.
ready	A task is *ready* when it is open, not paused, past any time constraints, and every task in its `blocked_by` list is terminal.
loop edge	A conditional back-edge (`loops_to`) that fires when its source task completes, re-opening a target task upstream. Not a blocking edge —does not affect readiness. Every loop edge has a mandatory `max_iterations` cap.
guard	A condition on a loop edge. Three kinds: *Always*, *TaskStatus*, and *IterationLessThan*.
loop iteration	A counter tracking how many times a task has been re-activated by a loop edge.
role	An agency entity defining *what* an agent does. Contains a description, skills, and a desired outcome. Identified by a content-hash of its identity-defining fields.
motivation	An agency entity defining *why* an agent acts the way it does. Contains a description, acceptable trade-offs, and unacceptable trade-offs. Identified by a content-hash of its identity-defining fields.

Term	Definition
agent	The unified identity in the agency system—a named pairing of a role and a motivation. Identified by a content-hash of `(role_id, motivation_id)`.
agency	The collective system of roles, motivations, and agents. Also refers to the storage directory (`.workgraph/agency/`).
content-hash ID	A SHA-256 hash of an entity's identity-defining fields. Deterministic, deduplicating, and immutable. Displayed as 8-character hex prefixes.
capability	A flat string tag on an agent used for task-to-agent matching at dispatch time. Distinct from role skills: capabilities are for *routing*, skills are for *prompt injection*.
skill	A capability reference attached to a role. Four types: *Name*, *File*, *Url*, *Inline*. Resolved at dispatch time and injected into the prompt.
trust level	A classification on an agent: *verified*, *provisional* (default), or *unknown*. Verified agents receive a small scoring bonus in task matching.
executor	The backend that runs an agent's work. Built-in: *claude* (AI), *shell* (automated command). Custom executors can be defined as TOML files.
coordinator	The scheduling brain inside the service daemon. Runs a tick loop that finds ready tasks and spawns agents.
service daemon	The background process started by `wg service start`. Hosts the coordinator, listens on a Unix socket for IPC, and manages agent lifecycle.
tick	One iteration of the coordinator loop. Triggered by IPC or a safety-net poll timer.
dispatch	The full cycle of selecting a ready task and spawning an agent: claim + spawn + register.
claim	Marking a task as *in-progress* and recording who is working on it. Distinct from *assignment*—claiming sets execution state.
assignment	Binding an agency agent identity to a task. Sets identity, not execution state.
auto-assign	A coordinator feature that creates `assign-{task-id}` meta-tasks for unassigned ready work.
auto-evaluate	A coordinator feature that creates `evaluate-{task-id}` meta-tasks for completed work.

Term	Definition
evaluation	A scored assessment of an agent's work. Four dimensions: correctness (40%), completeness (30%), efficiency (15%), style adherence (15%). Scores propagate to the agent, its role, and its motivation.
performance record	A running tally on each agent, role, and motivation: task count, average score, and evaluation references with context IDs.
evolution	The process of improving agency entities based on evaluation data. Triggered manually via `wg evolve`.
strategy	An evolution approach: *mutation*, *crossover*, *gap analysis*, *retirement*, *motivation tuning*, or *all*.
lineage	Evolutionary history on every role, motivation, and agent. Records parent IDs, generation number, creator identity, and timestamp.
generation	Steps from a manually-created ancestor. Generation 0 = human-created. Each evolution increments by one.
synergy matrix	A performance cross-reference of every (role, motivation) pair, showing average score and evaluation count.
meta-task	A task created by the coordinator to manage the agency loop. Assignment, evaluation, and evolution review tasks are meta-tasks.
map/reduce pattern	An emergent workflow: fan-out (one task unblocks parallel children) and fan-in (parallel tasks block a single aggregator). Arises from dependency edges, not a built-in primitive.
triage	An LLM-based assessment of a dead agent's output, classifying the result as *done*, *continue*, or *restart*.
wrapper script	The `run.sh` generated for each spawned agent. Runs the executor, captures output, and handles post-exit fallback logic.

2 System Overview

Workgraph is a task coordination system for humans and AI agents. It models work as a directed graph: tasks are nodes, dependency edges connect them, and a scheduler moves through the structure by finding what is ready and dispatching agents to do it. Everything—the graph, the agent identities, the configuration—lives in plain files under version control. There is no database. There is no mandatory server. The simplest possible deployment is a directory and a command-line tool.

But simplicity of storage belies richness of structure. The graph is not a flat list. Dependencies create ordering, parallelism emerges from independence, and loop edges introduce intentional cycles where work revisits earlier stages. Layered on top of this graph is an *agency*—a system of composable identities that gives each agent a declared purpose and a set of constraints. Together, the graph and the agency form a coordination system where the work is precisely defined, the workers are explicitly characterized, and improvement is built into the process.

This section establishes the big picture. The details follow in later sections: the task graph in Section 3, the agency model in Section 4, coordination and execution in Section 5, and evolution in Section 6.

2.1 The Graph Is the Work

A *task* is the fundamental unit of work in workgraph. Every task has an ID, a title, a status, and may carry metadata: estimated hours, required skills, deliverables, inputs, tags. Tasks are the atoms. Everything else—dependencies, scheduling, dispatch—is structure around them.

Tasks are connected by *dependency* edges expressed through the `blocked_by` field. If task B lists task A in its `blocked_by`, then B cannot begin until A reaches a *terminal* status—that is, until A is done, failed, or abandoned. This is a deliberate choice: all three terminal statuses unblock dependents, because a failed upstream task should not freeze the entire graph. The downstream task gets dispatched and can decide what to do with a failed predecessor.

From these simple rules, complex structures emerge. A single task blocking several children creates a fan-out pattern—parallel work radiating from a shared prerequisite. Several tasks blocking one aggregator create a fan-in—convergence into a synthesis step. Linear chains form pipelines. These are not built-in primitives. They arise naturally from dependency edges, the way sentences arise from words.

The graph is also not required to be acyclic. *Loop edges*—conditional back-edges declared through `loops_to`—allow a task to re-open an upstream task upon completion. A write-review-revise cycle, a CI retry pipeline, a monitoring loop: all are expressible as dependency chains with a loop edge pointing backward. Every loop edge carries a mandatory `max_iterations` cap and an optional *guard* condition. Loop edges are *not* dependency edges. They do not affect scheduling or readiness. They fire only when their source task completes, and only if the guard is satisfied and iterations remain. This keeps the scheduler's logic clean: it sees only the forward edges.

The entire graph lives in a single JSONL file—one JSON object per line, human-readable, friendly to version control, protected by file locking for concurrent writes. This is the canonical state. Every command reads from it; every mutation writes to it.

2.2 The Agency Is Who Does It

Without the agency system, every AI agent dispatched by workgraph is a blank slate—a generic assistant that receives a task description and does its best. This works, but it leaves performance

on the table. A generic agent has no declared priorities, no persistent personality, no way to improve across tasks. The agency system addresses this by giving agents *composable identities*.

An identity has two components. A *role* defines *what* the agent does: its description, its skills, its desired outcome. A *motivation* defines *why* the agent acts the way it does: its priorities, its acceptable trade-offs, and its hard constraints. The same role paired with different motivations produces different agents. A Programmer role with a Careful motivation—one that prioritizes reliability and rejects untested code—will behave differently than the same Programmer role with a Fast motivation that tolerates rough edges in exchange for speed. The combinatorial identity space is the key insight: a handful of roles and motivations yield a diverse population of agents.

Each role, each motivation, and each agent is identified by a *content-hash ID*—a SHA-256 hash of its identity-defining fields, displayed as an eight-character prefix. Content-hashing gives three properties that matter: identity is deterministic (same content always produces the same ID), deduplicating (you cannot create two identical entities), and immutable (changing an identity-defining field produces a *new* entity; the old one remains). This makes identity a mathematical fact, not an administrative convention. You can verify that two agents share the same role by comparing hashes.

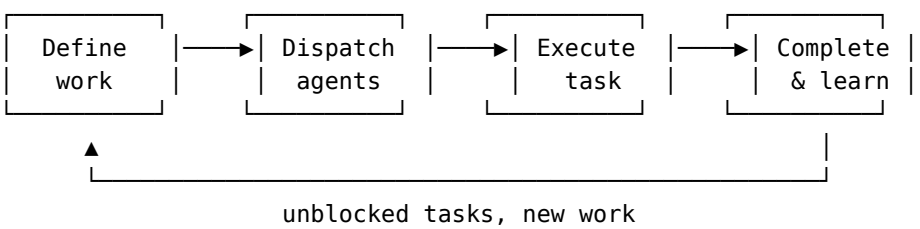
When an agent is dispatched to a task, its role and motivation are resolved—skills fetched from files, URLs, or inline definitions—and injected into the prompt. The agent doesn't just receive a task description; it receives an identity. This is what separates a workgraph agent from a one-off LLM call.

The agency as a whole—the collection of roles, motivations, and their pairings—functions as a *stable* of specialized workers. Each agent in the stable has a distinct behavioral signature derived from its role-motivation combination. You design the stable to match your project's needs: a few generalists for routine work, specialists for domain-specific tasks, careful reviewers, fast prototypers. The stable evolves over time as evaluation data reveals which identities perform well and which need improvement.

Human agents participate in the same model. The only difference is the *executor*: AI agents use `claude` (or another LLM backend); human agents use `matrix`, `email`, `shell`, or another human-facing channel. Human agents don't need roles or motivations—they bring their own judgment. But both human and AI agents are tracked, evaluated, and coordinated uniformly. The system does not distinguish between them in its bookkeeping; only the dispatch mechanism differs.

2.3 The Core Loop

Workgraph operates through a cycle that applies at every scale, from a single task to a multi-week project:



Listing 1: The heartbeat of a workgraph project.

Define work. Add tasks to the graph with their dependencies, skills, deliverables, and time estimates. The graph is the plan. Modifying it is cheap—add a task, change a dependency, split a bloated task into subtasks. The graph adapts as understanding evolves.

Dispatch agents. A *coordinator*—the scheduling brain inside an optional service daemon—finds *ready* tasks: those that are open, not paused, past any time constraints, and whose every dependency has reached a terminal status. For each ready task, it resolves the executor, builds context from completed dependencies, renders the prompt with the agent's identity, and spawns a detached process. The coordinator *claims* the task before spawning to prevent double-dispatch. Each agent instance handles exactly one task—one task, one process, one identity. When that task is done, the agent process exits. If more work is ready, the coordinator spawns a new agent for it.

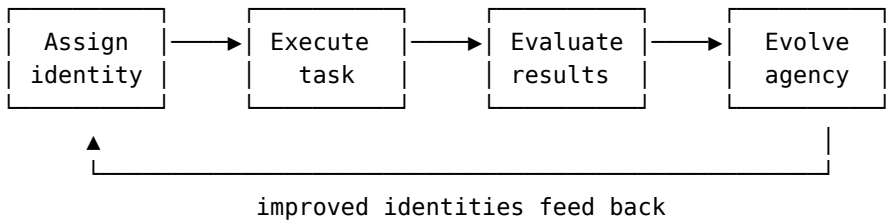
Execute. The spawned agent does its work. It may log progress, record artifacts, create subtasks, or mark the task done or failed. It operates with full autonomy within the boundaries set by its role and motivation.

Complete and learn. When a task reaches a terminal status, its dependents may become ready, continuing the flow. If the agency system is active, a completed task can also trigger *evaluation*—a scored assessment across four dimensions (correctness, completeness, efficiency, style adherence) whose results propagate to the agent, its role, and its motivation.

This is the basic heartbeat. Most projects run on this loop alone.

2.4 The Agency Loop

The agency system extends the core loop with a second, slower cycle of improvement:



Listing 2: The agency improvement cycle.

Assign identity. Before a task is dispatched, an agent identity is bound to it—either manually or through an auto-assign system where a dedicated assigner agent evaluates the available agents and picks the best fit. *Assignment* sets identity; it is distinct from *claiming*, which sets execution state.

Execute task. The agent works with its assigned identity injected into the prompt.

Evaluate results. After the task completes, an evaluator agent scores the work. Evaluation produces a weighted score that propagates to three levels: the agent, its role (with the motivation as context), and its motivation (with the role as context). This three-level propagation creates the data needed for cross-cutting analysis—how does a role perform with different motivations, and vice versa?

Evolve the agency. When enough evaluations accumulate, an evolver agent analyzes performance data and proposes structured changes: mutate a role to strengthen a weak dimension, cross two high-performing roles into a hybrid, retire a consistently poor motivation, create an entirely new role for unmet needs. Modified entities receive new content-hash IDs with *lineage* metadata linking them to their parents, creating an auditable evolutionary history.

These modified entities—mutated roles, tuned motivations, crossover hybrids—become the new generation of identities. Agents built from evolved components are themselves new agents with new content-hash IDs. Evolution is a manual trigger (`wg evolve`), not an automated process, because the human decides when there is enough data to act on and reviews every proposed change.

Each step in this cycle can be manual or automated. A project might start with manual assignment and no evaluation, graduate to auto-assign once agent identities stabilize, enable auto-evaluate to build a performance record, and eventually run evolution to refine the agency. The system meets you where you are.

2.5 How They Relate

The task graph and the agency are complementary systems with a clean separation. The graph defines *what* needs to happen and *in what order*. The agency defines *who* does it and *how they approach it*. Neither depends on the other for basic operation: you can run workgraph without the agency (every agent is generic), and you can define agency entities without a graph (though they have nothing to do). The power is in the combination.

The coordinator sits at the intersection. It reads the graph to find ready work, reads the agency to resolve agent identities, dispatches the work, and—when evaluation is enabled—closes the feedback loop by scoring results and feeding data back into the agency. The graph is the skeleton; the agency is the musculature; the coordinator is the nervous system.

Everything is files. The graph is JSONL. Agency entities—roles, motivations, agents—are YAML. Configuration is TOML. Evaluations are YAML. There is no database, no external dependency, no required network connection. The optional service daemon automates dispatch but is not required for operation. You can run the entire system from the command line, one task at a time, or you can start the daemon and let it manage a fleet of parallel agents. The architecture scales from a solo developer tracking personal tasks to a coordinated multi-agent project with dozens of concurrent workers, all from the same set of files in a `.workgraph` directory.

3 The Task Graph

Work is structure. A project without structure is a list—and lists lie. They hide the fact that you cannot deploy before you test, cannot test before you build, cannot build before you design. A list says "here are things to do." A graph says "here is the order in which reality permits you to do them."

Workgraph models work as a directed graph. Tasks are nodes. Dependencies are edges. The graph is the single source of truth for what exists, what depends on what, and what is available for execution right now. Everything else—the coordinator, the agency, the evolution system—reads from this graph and writes back to it. The graph is not a view of the project. It *is* the project.

3.1 Tasks as Nodes

A task is the atom of work. It has an identity, a lifecycle, and a body of metadata that guides both human and machine execution. Here is the anatomy:

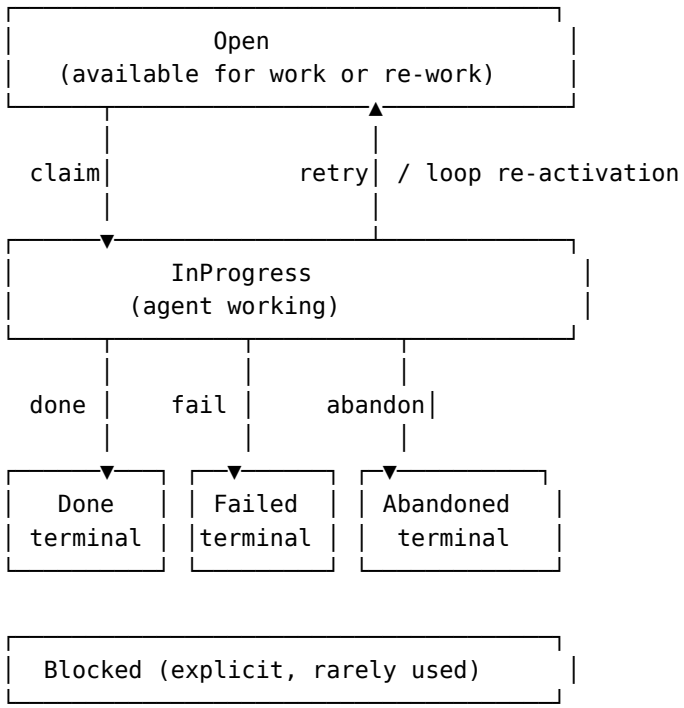
Field	Purpose
`id`	A slug derived from the title at creation time. The permanent key—used in every edge, every command, every reference. Once set, it never changes.
`title`	Human-readable name. Can be updated without breaking references.
`description`	The body: acceptance criteria, context, constraints. What an agent (human or AI) needs to understand the work.
`status`	Lifecycle state. One of six values—see below.
`estimate`	Optional cost and hours. Used by budget fitting and forecasting.
`tags`	Flat labels for filtering and grouping.
`skills`	Required capabilities—matched against agent capabilities at dispatch time.
`inputs`	Paths or references the task needs to read.
`deliverables`	Expected outputs—what the task should produce.
`artifacts`	Actual outputs recorded after completion.
`exec`	A shell command for automated execution via the shell executor.
`model`	Preferred AI model (haiku, sonnet, opus). Overrides coordinator and agent defaults.
`verify`	Verification criteria—if set, the task requires review before it can be marked done.
`agent`	Content-hash ID binding an agency agent identity to this task.
`log`	Append-only progress entries with timestamps and optional actor attribution.

Table 1: Task fields. Every field except `id`, `title`, and `status` is optional.

Tasks are not just descriptions of work—they are self-contained dispatch packets. An agent spawned for a task receives the description, the inputs, the skills, the log history, and the artifacts of completed dependencies. Everything needed to begin work is encoded on the node itself or reachable through its edges.

3.2 Status and Lifecycle

A task moves through six statuses. Most follow the happy path; some take detours.



Listing 3: Task state machine. The three terminal statuses share a critical property: they all unblock dependents.

Open is the starting state. A task is open when it has been created and is potentially available for work—though it may not yet be *ready* (a distinction explored below).

InProgress means an agent has claimed the task and is working on it. The coordinator sets this atomically before spawning the agent process.

Done, **Failed**, and **Abandoned** are the three *terminal* statuses. A terminal task will not progress further without explicit intervention—retry, manual re-open, or loop re-activation. The crucial design choice: all three terminal statuses unblock dependents. A failed upstream does not freeze the graph. The downstream task gets dispatched and can decide for itself what to do about a failed dependency—inspect the failure reason, skip the work, or adapt.

Blocked exists as an explicit status but is rarely used. In practice, blocking is *derived* from dependencies, not declared. A task whose `blocked_by` list contains non-terminal entries is blocked in the derived sense, regardless of its status field. The explicit `Blocked` status is a manual override for cases where a human wants to freeze a task for reasons outside the graph.

3.3 Terminal Statuses Unblock: A Design Choice

This merits emphasis. In many task systems, a failed dependency blocks everything downstream until a human intervenes. Workgraph takes the opposite stance: failure is information, not obstruction.

When task A fails and task B depends on A, B becomes ready. B's agent receives context from A—the failure reason, the log entries, the artifacts (if any). The agent can then decide: retry the work itself, produce a partial result, or fail explicitly with its own reason. The graph keeps moving.

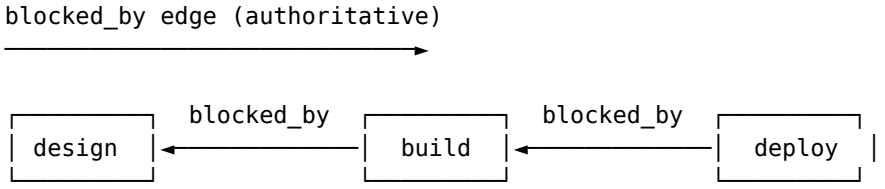
This works because terminal means "this task has reached an endpoint for this iteration." Done is a successful endpoint. Failed is an unsuccessful one. Abandoned is a deliberate withdrawal. In

all three cases, the task is no longer going to change, so dependents can proceed with whatever information is available.

The alternative—frozen pipelines waiting for human intervention—violates the principle that the graph should be self-advancing. If you need a hard stop on failure, model it explicitly: add a guard condition or a verification step. Don't rely on the scheduler to enforce business logic through status propagation.

3.4 Dependencies: `blocked_by` and `blocks`

Dependencies are directed edges. Task B depends on task A means: B cannot be ready until A reaches a terminal status. This is expressed by placing A's ID in B's `blocked_by` list.




```
Read as: build is blocked by design. deploy is blocked by build.
Equivalently: design blocks build. build blocks deploy.
```

Listing 4: Dependency edges. `blocked_by` is authoritative; `blocks` is its computed inverse.

The `blocked_by` list is the source of truth. The `blocks` list is its inverse, maintained for bidirectional traversal—if B is blocked by A, then A's `blocks` list includes B. The scheduler never reads `blocks`; it only checks `blocked_by`. The inverse is a convenience index for commands like `wg impact` and `wg bottlenecks` that need to traverse the graph forward from a task to its dependents.

Transitivity works naturally. If C is blocked by B and B is blocked by A, then C cannot be ready while A is non-terminal, because B cannot be ready (and thus cannot become terminal) while A is non-terminal. No transitive closure computation is needed—the scheduler checks each task's immediate blockers, and the chain resolves itself one link at a time.

A subtlety: if a task references a blocker that does not exist in the graph, the missing reference is treated as resolved. This is a fail-open design—a dangling reference does not freeze the graph. The `wg check` command flags these as warnings, but the scheduler proceeds.

3.5 Readiness

A task is *ready* when four conditions hold simultaneously:

1. **Open status.** The task must be in the `Open` state. Tasks that are in-progress, done, failed, abandoned, or explicitly blocked are never ready.
2. **Not paused.** The task's `paused` flag must be false. Pausing is an explicit hold—the task retains its status and all other state, but the coordinator will not dispatch it.
3. **Past time constraints.** If the task has a `not_before` timestamp, the current time must be past it. If the task has a `ready_after` timestamp (set by loop edge delays), the current time must be past that too. Invalid or missing timestamps are treated as satisfied—they do not block.
4. **All blockers terminal.** Every task ID in the `blocked_by` list must correspond to a task in a terminal status (done, failed, or abandoned). Non-existent blockers are treated as resolved.

These four conditions are evaluated by `ready_tasks()`, the function that the coordinator calls every tick to find work to dispatch. Ready is a precise, computed property—not a flag someone sets. You cannot manually mark a task as ready; you can only create the conditions under which the scheduler derives it.

The `not_before` field enables future scheduling: "do not start this task before next Monday." The `ready_after` field serves a different purpose—it is set automatically by loop edges with delays, creating pacing between loop iterations. Both are checked against the current wall-clock time.

3.6 Loop Edges: Intentional Cycles

Workgraph is a directed graph, not a DAG. This is a deliberate design choice.

Most task systems are acyclic by construction—dependencies flow in one direction, and cycles are errors. This works for projects that execute once: design, build, test, deploy, done. But real work is often iterative. You write a draft, a reviewer reads it, you revise based on feedback, the reviewer reads again. A CI pipeline builds, tests, and if tests fail, loops back to build with fixes. A monitoring system checks, investigates, fixes, verifies, and then checks again.

These patterns are cycles, and they are not bugs. They are the structure of iterative work. Workgraph makes them first-class through *loop edges*.

3.6.1 The `loops_to` Mechanism

A loop edge is a conditional back-edge declared via the `loops_to` field on a task. It says: "when I complete, evaluate a condition—if true and iterations remain, re-open the target task upstream."

Field	Purpose
`target`	The task ID to re-activate. Must be upstream (earlier in the dependency chain).
`guard`	A condition that must be true for the loop to fire. Optional—if absent, the loop fires unconditionally (up to `max_iterations`).
`max_iterations`	Hard cap on how many times the target can be re-activated by this edge. Mandatory—no unbounded loops.
`delay`	Optional duration (e.g., `"30s"`, `"5m"`, `"1h"`) to wait before the re-activated target becomes ready. Sets the target's `ready_after` timestamp.

Table 2: Loop edge fields. Every loop edge requires a target and a max_iterations cap.

The critical property: **loop edges are not blocking edges.** They are completely separate from `blocked_by`. They do not appear in the dependency lists. The scheduler never reads them when computing readiness. They exist only as post-completion triggers—evaluated when the source task transitions to done, and ignored at all other times.

This separation is the key insight of the design. The forward dependency chain (via `blocked_by`) remains acyclic and schedulable. The backward loop edge (via `loops_to`) layers iteration on top without disturbing the scheduler.

3.6.2 Guards

A guard is a condition on a loop edge that controls whether the loop fires. Three kinds:

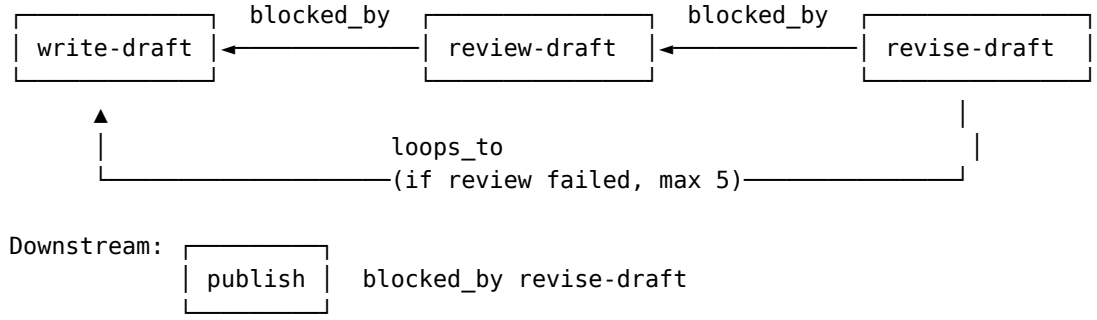
- **Always.** The loop fires unconditionally on every completion, up to `max_iterations`. Used for monitoring loops and fixed-iteration patterns.

- **TaskStatus.** The loop fires only if a named task has a specific status. The classic use: "loop back to writing if the review task failed." This is the mechanism for conditional retry.
- **IterationLessThan.** The loop fires only if the target's iteration count is below a threshold. Redundant with `max_iterations` in simple cases, but explicit when you want the guard condition visible in the graph data.

If no guard is specified, the loop behaves as `Always`—it fires on every completion up to the iteration cap.

3.6.3 A Review Loop, Step by Step

Consider a three-task review cycle:



Listing 5: A review loop. Forward edges (blocked_by) are solid. The loop edge is conditional.

The forward chain is acyclic: `write-draft` → `review-draft` → `revise-draft` → `publish`. The loop edge on `revise-draft` points back to `write-draft` with a guard: fire only if `review-draft` has status `Failed`, up to 5 iterations.

Here is the execution:

1. `write-draft` is open with no blockers—it is ready. The coordinator dispatches an agent.
2. The agent completes the draft and calls `wg done write-draft`. The task becomes terminal.
3. `review-draft` has all blockers terminal (just `write-draft`). It becomes ready. The coordinator dispatches a reviewer agent.
4. The reviewer finds problems and calls `wg fail review-draft --reason "Missing section 3"`. The task is now terminal (failed).
5. `revise-draft` has all blockers terminal (`review-draft` is failed—and failed is terminal). It becomes ready. The coordinator dispatches an agent.
6. The agent reads the failure reason from `review-draft`, revises accordingly, and calls `wg done revise-draft`.
7. On completion, the loop edge fires: the guard checks `review-draft`'s status—it is `Failed`. The iteration count on `write-draft` is 0, which is below `max_iterations` (5). The loop fires.
8. `write-draft` is re-opened: status set to `Open`, timestamps cleared, `loop_iteration` incremented to 1. A log entry records: "Re-activated by loop from revise-draft (iteration 1/5)."
9. `review-draft` is also re-opened—it is an intermediate task between the loop target (`write-draft`) and the loop source (`revise-draft`), and it was previously terminal.
10. `revise-draft` itself is re-opened—the source task re-enters the cycle.
11. `write-draft` is now open with no non-terminal blockers. The cycle begins again.

If the reviewer eventually approves (calls `wg done review-draft` instead of `wg fail`), then when `revise-draft` completes, the loop guard checks `review-draft`'s status—it is `Done`, not `Failed`.

The guard condition is not met. The loop does not fire. `revise-draft` stays done. `publish` has all blockers terminal. The graph proceeds.

3.6.4 Intermediate Re-Opening

When a loop fires and re-opens its target, the system also re-opens intermediate tasks—those on the dependency path between the target and the source that were previously terminal. This ensures the entire cycle is available for re-execution, not just the target.

The source task itself is also re-opened. It was just marked done by the agent, but it is part of the loop and must execute again in the next iteration. The system sets its status back to `Open`, clears its assignment and timestamps, and sets its `loop_iteration` to match the newly re-activated target.

Strictly speaking, intermediate tasks do not need explicit re-opening to be *correct*—the scheduler would not mark them ready anyway, because their blocker (the freshly re-opened target) is no longer terminal. But re-opening them explicitly ensures their status accurately reflects the loop state, and prevents them from appearing as "done" in status reports when they are actually pending re-execution.

3.6.5 Bounded Iteration

Every loop edge must specify `max_iterations`. There are no unbounded loops. When the target's `loop_iteration` reaches the cap, the loop edge stops firing, regardless of guard conditions. The task stays done. Downstream work proceeds.

This is a safety property. A guard condition with a logic error could fire indefinitely; `max_iterations` guarantees that every cycle terminates. The cap is per-edge—if multiple loop edges point at the same target, each has its own limit.

3.6.6 Loop Delays

A loop edge can specify a `delay`: a human-readable duration like `"30s"`, `"5m"`, `"1h"`, or `"1d"`. When a delayed loop fires, instead of making the target immediately ready, it sets the target's `ready_after` timestamp to `now + delay`. The scheduler will not dispatch the target until the delay has elapsed.

This creates pacing between iterations. A monitoring loop that checks system health every five minutes uses a delay of `"5m"`. A review loop that gives the author time to revise before the next review might use `"1h"`.

3.7 Pause and Resume

Sometimes you need to stop a loop—or any task—without destroying its state. The `paused` flag provides this control.

`wg pause <task>` sets the flag. The task retains its status, its loop iteration count, its log entries—everything. But the scheduler will not dispatch it. It is invisible to `ready_tasks()`.

`wg resume <task>` clears the flag. The task re-enters the readiness calculation. If it meets all four readiness conditions, it becomes available for dispatch on the next coordinator tick.

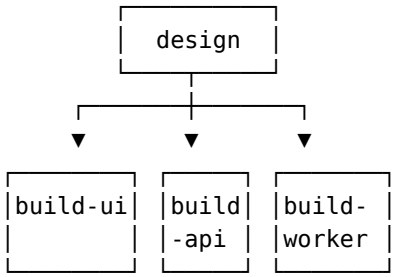
Pausing is orthogonal to status. You can pause an open task to hold it. You can pause a task mid-loop to halt the cycle without losing iteration state. When you resume, the loop picks up where it left off.

3.8 Emergent Patterns

The dependency graph and loop edges are the only primitives. But from these two mechanisms, several structural patterns emerge naturally.

3.8.1 Fan-Out (Map)

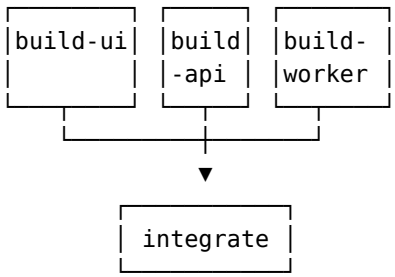
One task blocks several children. When the parent completes, all children become ready simultaneously and can execute in parallel.



Listing 6: Fan-out: one parent unblocks parallel children.

3.8.2 Fan-In (Reduce)

Several tasks block a single aggregator. The aggregator becomes ready only when all of its blockers are terminal.



Listing 7: Fan-in: multiple parents must all complete before the child is ready.

Combined, fan-out and fan-in produce the *map/reduce pattern*: a coordinator task fans out parallel work, then an aggregator task fans in the results. This is not a built-in primitive. It arises naturally from the shape of the dependency edges. The coordinator handles it without special configuration—`max_agents` (see Section 5) determines how many parallel tasks run concurrently.

3.8.3 Pipelines

A linear chain: A blocks B blocks C blocks D. Each task becomes ready only when its single predecessor completes. Pipelines are the simplest dependency structure—a sequence.

3.8.4 Review Loops

A forward chain with a loop edge, as described above. The cycle executes repeatedly until a guard condition breaks it or the iteration cap is reached. Review loops are the canonical example of intentional cycles.

3.9 Graph Analysis

Workgraph provides several analysis tools that read the graph structure and compute derived properties. These are instruments, not concepts—they report on the graph rather than define it.

Critical path. The longest dependency chain among active (non-terminal) tasks, measured in estimated hours. The critical path determines the minimum time to completion—no amount of parallelism can shorten it. Tasks on the critical path have zero slack; delays to any of them delay the entire project. `wg critical-path` computes this, skipping cycles to avoid infinite traversals.

Bottlenecks. Tasks that transitively block the most downstream work. A bottleneck is not necessarily on the critical path—it might block many short chains rather than one long one. `wg bottlenecks` ranks tasks by the count of transitive dependents, providing recommendations for tasks that should be prioritized.

Impact. Given a specific task, what depends on it? `wg impact <task>` traces both direct and transitive dependents, computing the total hours at risk if the task is delayed or fails.

Cost. The total estimated cost of a task including all its transitive dependencies, computed with cycle detection to avoid double-counting shared ancestors in diamond patterns.

Forecast. Projected completion date based on remaining work, estimated velocity, and dependency structure.

These tools share a common pattern: they traverse the graph using `blocked_by` edges (and their inverse), respect the visited-set pattern to handle cycles safely, and report on the structure without modifying it.

3.10 Storage

The graph is stored as JSONL—one JSON object per line, one node per object. A graph file might look like this:

```
{"kind":"task","id":"write-draft","title":"Write draft","status":"open","loops_to":
[]}
{"kind":"task","id":"review-draft","title":"Review
draft","status":"open","blocked_by":["write-draft"]}
{"kind":"task","id":"revise-draft","title":"Revise","status":"open","blocked_by":
["review-draft"],"loops_to":[{"target":"write-draft","guard":{"TaskStatus":
{"task":"review-draft","status":"failed"}},"max_iterations":5}]}
{"kind":"task","id":"publish","title":"Publish","status":"open","blocked_by":
["revise-draft"]}
```

Listing 8: A graph file in JSONL format. Each line is a self-contained node.

JSONL has three virtues for this purpose. It is human-readable—you can inspect and edit it with any text editor. It is version-control-friendly—adding or modifying a task changes one line, producing clean diffs. And it supports atomic writes with file locking—concurrent processes cannot corrupt the graph because every write acquires an exclusive lock, rewrites the file, and releases.

The graph file lives at `.workgraph/graph.jsonl` and is the canonical state of the project. There is no database, no server dependency. Everything reads from and writes to this file. The service daemon, when running, holds no state beyond what the file contains—it can be killed and restarted without loss.

—

The task graph is the foundation. Dependencies encode the ordering constraints of reality. Loop edges encode the iterative patterns of practice. Readiness is a derived property—the scheduler's answer to "what can happen next?" The coordinator uses this answer to dispatch work, as

described in Section 5. The agency system uses the graph to record evaluations at each task boundary, feeding the evolution process described in Section 6.

A well-designed task graph does not just organize work. It makes the structure of the project legible—to humans reviewing progress, to agents receiving dispatch, and to the system itself as it learns from its own history.

4 The Agency Model

A generic AI assistant is a blank slate. It has no declared priorities, no persistent personality, no way to accumulate craft. Every session starts from zero. The agency system exists to change this. It gives agents *composable identities*—a role that defines what the agent does, paired with a motivation that defines why it acts the way it does. The same role combined with a different motivation produces a different agent. This is the key insight: identity is not a name tag, it is a *function*—the Cartesian product of competence and intent.

The result is an identity space that grows combinatorially. Four roles and four motivations yield sixteen distinct agents, each with its own behavioral signature. These identities are not administrative labels. They are content-hashed, immutable, evaluable, and evolvable. An agent's identity is a mathematical fact, verifiable by anyone who knows the hash.

4.1 Roles

A role answers a single question: *what does this agent do?*

It carries three identity-defining fields:

- **Description.** A prose statement of the role's purpose—what kind of work it performs, what domain it operates in, what skills it brings to bear.
- **Skills.** A list of skill references that define the role's capabilities. These are resolved at dispatch time and injected into the agent's prompt as concrete instructions (see Section 4.5 below).
- **Desired outcome.** What good output looks like. This is the standard against which the agent's work will be evaluated—not a vague aspiration, but a crisp definition of success.

A role also carries a *name* (a human-readable label like "Programmer" or "Architect"), a *performance* record (aggregated evaluation scores), and *lineage* metadata (evolutionary history). These are mutable—they can change without altering the role's identity. The name is for humans. The identity is for the system.

Consider two roles: one describes a code reviewer who checks for correctness, testing gaps, and style violations; the other describes an architect who evaluates structural decisions and dependency management. They may share some skills, but their descriptions and desired outcomes differ, so they produce different content-hash IDs—different identities, different agents, different behaviors when paired with the same motivation.

4.2 Motivations

A motivation answers the complementary question: *why does this agent act the way it does?*

Where a role defines competence, a motivation defines character. It carries three identity-defining fields:

- **Description.** What this motivation prioritizes—the values and principles that guide the agent's approach to work.
- **Acceptable trade-offs.** Compromises the agent may make. A "Fast" motivation might accept less thorough documentation. A "Careful" motivation might accept slower delivery. These are the negotiable costs of the agent's operating philosophy.

- **Unacceptable trade-offs.** Hard constraints the agent must never violate. A "Careful" motivation might refuse to ship untested code under any circumstances. A "Thorough" motivation might refuse to skip edge cases. These are non-negotiable.

Like roles, motivations carry a mutable name, performance record, and lineage. And like roles, only the identity-defining fields contribute to the content-hash.

The distinction between acceptable and unacceptable trade-offs is not decorative. When an agent's identity is rendered into a prompt, the acceptable trade-offs appear as *operational parameters*—flexibility the agent may exercise—and the unacceptable trade-offs appear as *non-negotiable constraints*—lines it must not cross. The motivation shapes behavior through the prompt: same role, different motivation, different output.

4.3 Agents: The Pairing

An agent is the unified identity in the agency system. For AI agents, it is the named pairing of exactly one role and exactly one motivation:

$$\textbf{agent} = \textbf{role} \times \textbf{motivation}$$

The agent's content-hash ID is computed from `(role_id, motivation_id)`. Nothing else enters the hash. This means the agent is entirely determined by its constituents: if you know the role and the motivation, you know the agent.

An agent also carries operational fields that do not affect its identity:

Capabilities Flat string tags (e.g., `"rust"`, `"testing"`) used for task-to-agent matching at dispatch time. Capabilities are distinct from role skills: capabilities are for *routing* (which agent gets which task), skills are for *prompt injection* (what instructions the agent receives). An agent might have capabilities broader than its role's skills, or narrower, depending on how the operator configures it.

Rate An hourly rate for cost forecasting.

Capacity Maximum concurrent tasks this agent can handle.

Trust level A classification that affects dispatch priority (see Section 4.6 below).

Contact Email, Matrix ID, or other contact information—primarily for human agents.

Executor The backend that runs the agent's work (see Section 4.7 below).

The compositional nature of agents is what makes the identity space powerful. A "Programmer" role paired with a "Careful" motivation produces an agent that writes methodical, well-tested code and refuses to ship without tests. The same "Programmer" role paired with a "Fast" motivation produces an agent that prioritizes rapid delivery and accepts less thorough documentation. Both are programmers. They differ in *why* they program the way they do.

This is not a theoretical nicety. When the coordinator dispatches a task (see Section 5 for the full dispatch cycle), the agent's full identity—role description, skills, desired outcome, acceptable trade-offs, non-negotiable constraints—is rendered into the prompt. The AI receives a complete behavioral specification before it sees the task. The motivation is not a hint; it is a contract.

4.4 Content-Hash IDs

Every role, motivation, and agent is identified by a SHA-256 hash of its identity-defining fields. The hash is computed from canonical YAML serialization of those fields, ensuring determinism across platforms and implementations.

Entity	**Hashed fields**
Role	description + skills + desired outcome
Motivation	description + acceptable trade-offs + unacceptable trade-offs
Agent	role ID + motivation ID

Table 3: Identity-defining fields for content-hash computation.

Three properties follow from content-hashing:

Deterministic. The same content always produces the same ID. If two people independently create a role with identical description, skills, and desired outcome, they get the same hash. There is no ambiguity, no namespace collision, no registration authority.

Deduplicating. You cannot create two entities with identical identity-defining fields. The system detects the collision and rejects the duplicate. This is not a bug—it is a feature. If two roles are identical in what they do, they *are* the same role. The name might differ, but the identity does not.

Immutable. Changing any identity-defining field produces a *new* entity with a new hash. The old entity remains untouched. This means you never "edit" an identity—you create a successor. The original is preserved, its performance history intact, its lineage available for inspection. Mutable fields (name, performance, lineage) can change freely without affecting the hash.

For display, IDs are shown as 8-character hexadecimal prefixes (e.g., `a3f7c21d`). All commands accept unique prefixes—you type as few characters as needed to disambiguate.

Why does this matter? Content-hashing makes identity a verifiable fact. You can confirm that two agents share the same role by comparing eight characters. You can trace an agent's lineage through a chain of hashes, each linking to its parent. You can deduplicate across teams: if your colleague created the same role, the system knows. And because identity is immutable, performance data attached to a hash is *permanently* associated with a specific behavioral definition. A role's score of 0.85 means something precise—it is the score of *that exact* description, *those exact* skills, *that exact* desired outcome.

4.5 The Skill System

Skills are capability references attached to a role. They serve double duty: they declare what the role can do (for humans reading the role definition), and they inject concrete instructions into the agent's prompt (for the AI receiving the dispatch).

Four reference types exist:

Name A bare string label. `"rust"`, `"testing"`, `"architecture"`. No content beyond the tag itself. Used when the skill is self-explanatory and needs no elaboration — the word *is* the instruction.

File A path to a document on disk. The file is read at dispatch time and its full content is injected into the prompt. Supports absolute paths, relative paths (resolved from the project

root), and tilde expansion. Use this for project-specific style guides, coding standards, or domain knowledge that lives alongside the codebase.

Url An HTTP address. The content is fetched at dispatch time. Use this for shared resources that multiple projects reference—team-wide checklists, organization standards, living documents.

Inline Content embedded directly in the skill definition. The text is injected verbatim into the prompt. Use this for short, self-contained instructions: `"Write in a clear, technical style"` or `"Always include error handling for network calls"`.

Skill resolution happens at dispatch time. Name skills pass through as labels. File skills read from disk. Url skills fetch over HTTP. Inline skills use their embedded text. If a resolution fails—a file is missing, a URL is unreachable—the system logs a warning but does not block execution. The agent is spawned with whatever skills resolved successfully.

The distinction between role skills and agent capabilities is worth emphasizing. *Skills* are prompt content—they are instructions injected into the AI's context. *Capabilities* are routing tags—they are flat strings compared against a task's required skills to determine which agent is a good match. An agent's capabilities might list `"rust"` and `"testing"` for routing purposes, while its role's skills include a detailed Rust style guide (as a File reference) and a testing checklist (as Inline content). The routing system sees tags; the agent sees documents.

4.6 Trust Levels

Every agent carries a trust level: one of **Verified**, **Provisional**, or **Unknown**.

Verified Fully trusted. The agent has a track record or has been explicitly vouched for. Verified agents receive a small scoring bonus in task-to-agent matching, making them more likely to be dispatched for contested work.

Provisional The default for newly created agents. Neither trusted nor distrusted. Most agents start here and stay here unless explicitly promoted.

Unknown External or unverified. An agent from outside the team, or one that has not yet demonstrated reliability. Unknown agents receive no penalty—they simply lack the bonus that Verified agents enjoy.

Trust is set at agent creation time and can be changed later. It does not affect the agent's content-hash ID—trust is an operational classification, not an identity property.

4.7 Human and AI Agents

The agency system does not distinguish between human and AI agents at the identity level. Both are entries in the same agent registry. Both can have roles, motivations, capabilities, and trust levels. Both are tracked, evaluated, and appear in the synergy matrix. The identity model is uniform.

The difference is the **executor**—the backend that delivers work to the agent.

`claude` The default. Pipes a rendered prompt into the Claude CLI. The agent is an AI. Its role and motivation are injected into the prompt, shaping behavior through language.

`matrix` Sends a notification via the Matrix protocol. The agent is a human who monitors a Matrix room.

`email` Sends a notification via email. The agent is a human who checks their inbox.

`shell` Runs a shell command from the task's `exec` field. The agent is a human (or a script) that responds to a trigger.

For AI agents, role and motivation are *required*—an AI without identity is a blank slate, which is precisely what the agency system exists to prevent. For human agents, role and motivation are *optional*. Humans bring their own judgment, priorities, and character. A human agent might have a role (to signal what kind of work to route to them) or might operate without one (receiving any work that matches their capabilities).

Both types are evaluated using the same rubric. But human agent evaluations are excluded from the evolution signal—the system does not attempt to "improve" humans through the evolutionary process. Evolution operates only on AI identities, where changing the role or motivation has a direct, mechanistic effect on behavior through prompt injection.

4.8 Composition in Practice

To make the compositional nature of agents concrete, consider a small agency seeded with `wg agency init`. This creates four starter roles and four starter motivations:

Starter Roles	**Starter Motivations**
Programmer	Careful
Reviewer	Fast
Documenter	Thorough
Architect	Balanced

Table 4: The sixteen possible pairings from four roles and four motivations.

A "Programmer" paired with "Careful" produces an agent that writes methodical, tested code and treats untested output as a hard constraint violation. The same "Programmer" paired with "Fast" produces an agent that ships quickly and accepts less documentation as a reasonable trade-off. A "Reviewer" with "Thorough" examines every edge case and refuses to approve incomplete coverage. A "Reviewer" with "Balanced" weighs thoroughness against schedule pressure and accepts pragmatic compromises.

Each of these sixteen pairings has a unique content-hash ID. Each accumulates its own performance history. Over time, the evaluation data reveals which combinations excel at which kinds of work—the synergy matrix (Section 6.2.1) makes this visible. High-performing pairs are dispatched more often. Low-performing pairs are candidates for evolution or retirement.

The same compositionality applies to evolved entities. When the evolver mutates a role — say, refining the "Programmer" description to emphasize error handling—a *new* role is created with a new hash. Every agent that referenced the old role continues to exist unchanged. New agents can be created pairing the refined role with existing motivations. The old and new coexist, each with their own performance records, until the evidence shows which is superior.

4.9 Lineage and Deduplication

Content-hash IDs enable two properties that matter at scale: lineage tracking and deduplication.

Lineage. Every role, motivation, and agent records its evolutionary ancestry. A manually created entity has no parents and generation zero. A mutated entity records one parent and increments the generation. A crossover entity records two parents and increments from the

highest. The `created_by` field distinguishes human creation (`"human"`) from evolutionary creation (`"evolver-{run_id}"`).

Because identity is content-hashed, lineage is unfalsifiable. The parent entity cannot be silently altered—any change would produce a different hash, breaking the lineage link. You can walk the ancestry chain from any entity back to its manually created roots, confident that each link refers to the exact content that existed at creation time. This is not a version history in the traditional sense. It is an immutable record of how the agency's identity space has evolved.

Deduplication. If the evolver proposes a role that is identical to an existing one — same description, same skills, same desired outcome—the content-hash collision is detected and the duplicate is rejected. This prevents the agency from accumulating redundant entities. It also means that convergent evolution is recognized: if two independent mutation paths arrive at the same role definition, the system knows they are the same role.

5 Coordination & Execution

When you type `wg service start --max-agents 5`, a background process wakes up, binds a Unix socket, and begins to breathe. Every few seconds it opens the graph file, scans for ready tasks, and decides what to do. This is the coordinator—the scheduling brain that turns a static directed graph into a running system. Without it, workgraph is a notebook. With it, workgraph is a machine.

This section walks through the full lifecycle of work: from the moment the daemon starts, through the dispatch of agents, to the handling of their success, failure, and unexpected death.

5.1 The Service Daemon

The service daemon is a background process that hosts the coordinator, listens on a Unix socket for commands, and manages agent lifecycle. It is started with `wg service start` and stopped with `wg service stop`. Between those two moments it runs a loop: accept connections, process IPC requests, and periodically run the coordinator tick.

The daemon writes its PID and socket path to `.workgraph/service/state.json`—a lockfile of sorts. When you run `wg service status`, the CLI reads this file, checks whether the PID is alive, and reports the result. If the daemon crashes and leaves a stale state file, the next `wg service start` detects the dead PID, cleans up, and starts fresh. If you want to be forceful about it, `wg service start --force` kills any existing daemon before launching a new one.

All daemon activity is logged to `.workgraph/service/daemon.log`, a timestamped file with automatic rotation at 10 MB. The log captures every coordinator tick, every spawn, every dead agent detection, every IPC request. When something goes wrong, the answer is almost always in this file.

One detail matters more than it might seem: agents spawned by the daemon are *detached*. The spawn code calls `setsid()` to place each agent in its own session and process group. This means agents survive daemon restarts. You can stop the daemon, reconfigure it, start it again, and every running agent continues undisturbed. The daemon does not own its agents—it launches them and watches them from a distance.

5.2 The Coordinator Tick

The coordinator's heartbeat is the *tick*—a single pass through the scheduling logic. Two things trigger ticks: IPC events (immediate, reactive) and a background poll timer (a safety net that catches manual edits to the graph file). The poll interval defaults to 60 seconds and is configurable via `config.toml` or `wg service reload --poll-interval N`.

Each tick has six phases:

1. **Reap zombies.** Even though agents run in their own sessions, they remain children of the daemon process. When an agent exits, it becomes a zombie until the parent calls `waitpid`. The tick begins by reaping all zombies so that subsequent PID checks return accurate results.
2. **Clean up dead agents and count slots.** The coordinator walks the agent registry and checks each alive agent's PID. If the process is gone, the agent is dead. Dead agents have their tasks unclaimed—the task status reverts to open, ready for re-dispatch. The coordinator then counts truly alive agents (not just registry entries, but processes with running PIDs) and compares against `max_agents`. If all slots are full, the tick ends early.
3. **Build auto-assign meta-tasks.** If `auto_assign` is enabled in the agency configuration, the coordinator scans for ready tasks that have no agent identity bound to them. For each,

it creates an `assign-{task-id}` meta-task that blocks the original. This meta-task, when dispatched, will spawn an assigner agent that inspects the agency's roster and picks the best fit. The meta-task is tagged `"assignment"` to prevent recursive auto-assignment—the coordinator never creates an assignment task for an assignment task.

4. **Build auto-evaluate meta-tasks.** If `auto_evaluate` is enabled, the coordinator creates `evaluate-{task-id}` meta-tasks blocked by each work task. When the work task reaches a terminal status, the evaluation task becomes ready. Evaluation tasks use the shell executor to run `wg evaluate`, which spawns a separate evaluator to score the work. Tasks assigned to human agents are skipped—the system does not presume to evaluate human judgment. Meta-tasks tagged `"evaluation"`, `"assignment"`, or `"evolution"` are excluded to prevent infinite regress.

5. **Save graph and find ready tasks.** If the auto-assign or auto-evaluate phases modified the graph (adding meta-tasks, adjusting blockers), the coordinator saves it before proceeding. Then it computes the set of ready tasks. If no tasks are ready, the tick ends. If all tasks in the graph are terminal, the coordinator logs that the project is complete.

6. **Spawn agents.** For each ready task, up to the number of available slots, the coordinator dispatches an agent. This is where the dispatch cycle—the core of the system—begins.

```
┌──────────────────────────────────────────────┐
│                  TICK LOOP                   │
│                                              │
│  1. reap_zombies()                           │
│  2. cleanup_dead_agents → count alive slots  │
│  3. build_auto_assign_tasks    (if enabled)  │
│  4. build_auto_evaluate_tasks  (if enabled)  │
│  5. save graph → find ready tasks            │
│  6. spawn_agents_for_ready_tasks(slots_available) │
│                                              │
│  Triggered by: IPC graph_changed | poll timer │
└──────────────────────────────────────────────┘
```

Listing 9: The six phases of a coordinator tick.

5.3 The Dispatch Cycle

Dispatch is the act of selecting a ready task and spawning an agent for it. It is not a single operation but a sequence with careful ordering, because the coordinator must prevent double-dispatch: two ticks must never spawn two agents on the same task.

For each ready task, the coordinator proceeds as follows:

Resolve the executor. If the task has an `exec` field (a shell command), the executor is `shell`—no AI agent needed. Otherwise, the coordinator checks whether the task has an assigned agent identity. If it does, it looks up that agent's `executor` field (which might be `claude`, `shell`, or a custom executor). If no agent is assigned, the coordinator falls back to the service-level default executor (typically `claude`).

Resolve the model. Model selection follows a priority chain: the task's own `model` field takes precedence, then the coordinator's configured model, then the agent identity's model preference. This lets you pin specific tasks to specific models—a cheap model for routine evaluation tasks, a capable one for complex implementation.

Build context from dependencies. The coordinator reads each terminal dependency's artifacts (file paths recorded by the previous agent) and recent log entries. This context is injected into the prompt so the new agent knows what upstream work produced and what decisions were made. The agent does not start from a blank slate—it inherits the trail of work that came before it.

Render the prompt. The executor's prompt template is filled with template variables: `{{task_id}}`, `{{task_title}}`, `{{task_description}}`, `{{task_context}}`, `{{task_identity}}`. The identity block—the agent's role, motivation, skills, and operational parameters—comes from resolving the assigned agent's role and motivation from agency storage (see Section 4). Skills are resolved at this point: file skills read from disk, URL skills fetch via HTTP, inline skills expand in place. The rendered prompt is written to a file in the agent's output directory.

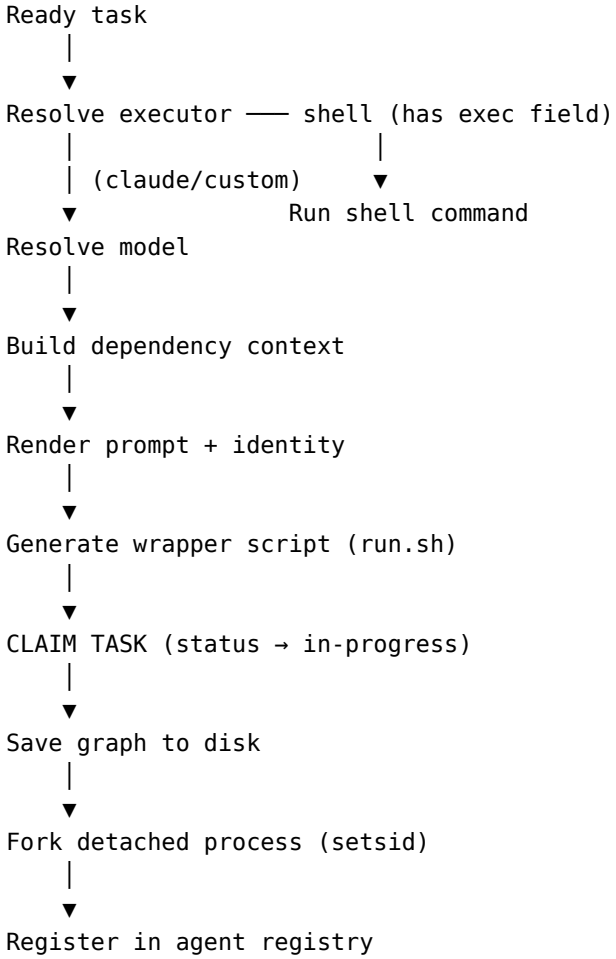
Generate the wrapper script. The coordinator writes a `run.sh` that:

- Unsets `CLAUDECODE` and `CLAUDE_CODE_ENTRYPOINT` environment variables so the spawned agent starts a clean session.
- Pipes the prompt file into the executor command (e.g., `cat prompt.txt | claude --print --verbose --output-format stream-json`).
- Captures all output to `output.log`.
- After the executor exits, checks whether the task is still in-progress. If the agent already called `wg done` or `wg fail`, the wrapper does nothing. If the task is still in-progress and the executor exited cleanly, the wrapper calls `wg done`. If it exited with an error, the wrapper calls `wg fail`. This safety net ensures tasks never get stuck in-progress after an agent dies silently.

Claim the task. Before spawning the process, the coordinator atomically sets the task's status to in-progress and records the agent ID in the `assigned` field. The graph is saved to disk at this point. If two coordinators somehow ran simultaneously, the second would find the task already claimed and skip it. The ordering is deliberate: claim first, spawn second. If the spawn fails, the coordinator rolls back the claim—reopening the task so it can be dispatched again.

Fork the detached process. The wrapper script is launched via `bash run.sh` with stdin, stdout, and stderr redirected. The `setsid()` call places the agent in its own session. The coordinator records the PID in the agent registry.

Register in the agent registry. The agent registry (`.workgraph/agents/registry.json`) tracks every spawned agent: ID, PID, task, executor, start time, heartbeat, status. The coordinator uses this registry to monitor agents across ticks.



Listing 10: The dispatch cycle, from ready task to running agent.

5.4 The Wrapper Script

The wrapper script deserves its own discussion because it solves a subtle problem: what happens when an agent dies without reporting its status?

An agent is expected to call `wg done <task-id>` when it finishes or `wg fail <task-id> --reason "..."` when it cannot complete the work. But agents crash. They get OOM-killed. Their SSH connections drop. The Claude CLI segfaults. In all these cases, the task would remain in-progress forever without the wrapper.

The wrapper runs the executor command, captures its exit code, then checks the task's current status via `wg show`. If the task is still in-progress—meaning the agent never called `wg done` or `wg fail`—the wrapper steps in. A clean exit (code 0) triggers `wg done`; a non-zero exit triggers `wg fail` with the exit code as the reason.

This two-layer design (agent self-reports, wrapper as fallback) means the system tolerates both well-behaved and badly-behaved agents. A good agent calls `wg done` partway through the wrapper execution, and when the wrapper later checks, it finds the task already done and does nothing. A crashing agent leaves the task in-progress, and the wrapper picks up the pieces.

5.5 Parallelism Control

The `max_agents` parameter is the single throttle on concurrency. When you start the service with `--max-agents 5`, the coordinator will never have more than five agents running simultaneously.

Each tick counts truly alive agents (verifying PIDs, not just trusting the registry) and only spawns into available slots.

This is a global cap, not per-task. Five agents might all be working on independent tasks in a fan-out pattern, or they might be serialized through a linear chain with only one active at a time. The coordinator does not reason about the graph's topology when deciding how many agents to spawn—it simply fills available slots with ready tasks, first-come-first-served.

You can change `max_agents` without restarting the daemon. `wg service reload --max-agents 10` sends a `Reconfigure` IPC message; the coordinator picks up the new value on the next tick. This lets you scale up when a fan-out creates many parallel tasks, then scale back down when work converges.

5.5.1 Map/Reduce Patterns

Parallelism in workgraph arises naturally from the graph structure. A *fan-out* (map) pattern occurs when one task blocks several children: the parent completes, all children become ready simultaneously, and the coordinator spawns agents for each (up to `max_agents`). A *fan-in* (reduce) pattern occurs when several tasks block a single aggregator: the aggregator only becomes ready when all its dependencies are terminal, and then a single agent handles the synthesis.

These patterns are not built-in primitives. They emerge from dependency edges, as described in Section 3. A project plan that says "write five sections, then compile the manual" naturally produces a fan-out of five writer tasks followed by a fan-in to a compiler task. The coordinator handles this without any special configuration—`max_agents` determines how many of the five writers run concurrently.

5.6 Auto-Assign

When the agency system is active and `auto_assign` is enabled in configuration, the coordinator automates the binding of agent identities to tasks. Without auto-assign, a human must run `wg assign <task-id> <agent-hash>` for each task. With it, the coordinator handles matching.

The mechanism is indirect. The coordinator does not contain matching logic itself. Instead, it creates a blocking `assign-{task-id}` meta-task for each unassigned ready task. This meta-task is dispatched like any other—an assigner agent (itself an agency entity with its own role and motivation) is spawned to evaluate the available agents and pick the best fit. The assigner reads the agency roster via `wg agent list`, compares capabilities to task requirements, considers performance history, and calls `wg assign <task-id> <agent-hash>` followed by `wg done assign-{task-id}`.

The result is a two-phase dispatch: first the assigner runs, binding an identity to the task. The assignment task completes, unblocking the original task. On the next tick, the original task is ready again—now with an agent identity attached—and the coordinator dispatches it normally.

Meta-tasks tagged `"assignment"`, `"evaluation"`, or `"evolution"` are excluded from auto-assignment. This prevents the coordinator from creating an assignment task for an assignment task, which would recurse infinitely.

5.7 Auto-Evaluate

When `auto_evaluate` is enabled, the coordinator creates evaluation meta-tasks for completed work. For every non-meta-task in the graph, an `evaluate-{task-id}` task is created, blocked by

the original. When the original task reaches a terminal status (done or failed), the evaluation task becomes ready and is dispatched.

Evaluation tasks use the shell executor to run `wg evaluate <task-id>`, which spawns a separate evaluator that reads the task definition, artifacts, and output logs, then scores the work on four dimensions: correctness (40% weight), completeness (30%), efficiency (15%), and style adherence (15%). The scores propagate to the agent, its role, and its motivation, building the performance data that drives the evolution process described in Section 6.

Two exclusions apply. Tasks assigned to human agents are not auto-evaluated—the system does not presume to score human work. And tasks that are themselves meta-tasks (tagged `"evaluation"`, `"assignment"`, or `"evolution"`) are excluded to prevent evaluation of evaluations.

Failed tasks also get evaluated. When a task's status is failed, the coordinator removes the blocker from the evaluation task so it becomes ready immediately. This is deliberate: failure modes carry signal. An agent that fails consistently on certain kinds of tasks reveals information about its role-motivation pairing that the evolution system can act on.

5.8 Dead Agent Detection and Triage

Every tick, the coordinator checks whether each agent's process is still alive. A dead agent—one whose PID no longer exists—triggers cleanup: the agent's task is unclaimed (status reverts to open), and the agent is marked dead in the registry.

But simple restart is wasteful when the agent made significant progress before dying. This is where *triage* comes in.

When `auto_triage` is enabled in the agency configuration, the coordinator does not immediately unclaim a dead agent's task. Instead, it reads the agent's output log and sends it to a fast, cheap LLM (defaulting to Haiku) with a structured prompt. The triage model classifies the result into one of three verdicts:

- **Done.** The work appears complete—the agent just didn't call `wg done` before dying. The task is marked done, and loop edges are evaluated.
- **Continue.** Significant progress was made. The task is reopened with recovery context injected into its description: a summary of what was accomplished, with instructions to continue from where the previous agent left off rather than starting over.
- **Restart.** Little or no meaningful progress. The task is reopened cleanly for a fresh attempt.

Both "continue" and "restart" respect `max_retries`. If the retry count exceeds the limit, the task is marked failed rather than reopened. The triage model runs synchronously with a configurable timeout (default 30 seconds), so it does not block the coordinator for long.

This three-way classification turns agent death from a binary event (restart or give up) into a nuanced recovery mechanism. A task that was 90% complete when the agent was OOM-killed does not lose its progress.

5.9 IPC Protocol

The daemon listens on a Unix socket (`.workgraph/service/daemon.sock`) for JSON-line commands. Every CLI command that modifies the graph—`wg add`, `wg done`, `wg fail`, `wg retry`—automatically sends a `graph_changed` message to wake the coordinator for an immediate tick.

The full set of IPC commands:

Command	Effect
`graph_changed`	Schedules an immediate coordinator tick. The fast path for reactive dispatch.
`spawn`	Directly spawns an agent for a specific task, bypassing the coordinator's scheduling.
`agents`	Returns the list of all registered agents with their status, PID, and uptime.
`kill`	Terminates a running agent by PID (graceful SIGTERM, then SIGKILL if forced).
`status`	Returns the coordinator's current state: tick count, agents alive, tasks ready.
`shutdown`	Stops the daemon. Running agents continue independently by default; `kill_agents` terminates them.
`pause`	Suspends the coordinator. No new agents are spawned, but running agents continue.
`resume`	Resumes the coordinator and triggers an immediate tick.
`reconfigure`	Updates `max_agents`, `executor`, `poll_interval`, or `model` at runtime without restart.
`heartbeat`	Records a heartbeat for an agent (used for liveness tracking).

The `reconfigure` command is particularly useful for live tuning. If a fan-out creates twenty parallel tasks and you only have five slots, you can bump `max_agents` to ten without stopping anything. When the fan-out completes and work converges, scale back down.

5.10 Custom Executors

Executors are defined as TOML files in `.workgraph/executors/`. Each specifies a command, arguments, environment variables, a prompt template, a working directory, and an optional timeout. The default `claude` executor pipes a prompt file into the Claude CLI with `--print` and `--output-format stream-json`. The default `shell` executor runs a bash command from the task's `exec` field.

Custom executors enable integration with any tool. An executor for a different LLM provider, a code execution sandbox, a notification system—any process that can be launched from a shell command can serve as an executor. The prompt template supports the same `{{task_id}}`, `{{task_title}}`, `{{task_description}}`, `{{task_context}}`, and `{{task_identity}}` variables as the built-in executors.

The executor also determines whether an agent is AI or human. The `claude` executor means AI. Executors like `matrix` or `email` (for sending notifications to humans) mean human. This distinction matters for auto-evaluation: human-agent tasks are skipped.

5.11 Pause, Resume, and Manual Control

The coordinator can be paused via `wg service pause`. In the paused state, no new agents are spawned, but running agents continue their work. This is useful when you need to make manual graph edits without the coordinator racing to dispatch tasks you are still arranging.

`wg service resume` lifts the pause and triggers an immediate tick.

For debugging and testing, `wg service tick` runs a single coordinator tick without the daemon. This lets you step through the scheduling logic one tick at a time, observing what the coordinator would do. And `wg spawn <task-id> --executor claude` dispatches a single task manually, bypassing the daemon entirely.

5.12 The Full Picture

Here is what happens, end to end, when a human operator types `wg service start --max-agents 5` on a project with tasks and an agency:

The daemon forks into the background. It opens a Unix socket, reads `config.toml` for coordinator settings, and writes its PID to the state file. Its first tick runs immediately.

The tick reaps zombies (there are none yet), checks the agent registry (empty), and counts zero alive agents out of a maximum of five. If `auto_assign` is enabled, it scans for ready tasks without agent identities and creates assignment meta-tasks. If `auto_evaluate` is enabled, it creates evaluation tasks for work tasks. It saves the graph if modified, then finds ready tasks.

Suppose three tasks are ready: two assignment meta-tasks and one task that was already assigned. The coordinator spawns three agents (five slots available, three tasks ready). Each spawn follows the dispatch cycle: resolve executor, resolve model, build context, render prompt, write wrapper script, claim task, fork process, register agent.

The three agents run concurrently. The two assigners examine the agency roster and bind identities. They call `wg done assign-{task-id}`, which triggers `graph_changed` IPC. The daemon wakes for an immediate tick. Now the two originally-unassigned tasks are ready (their assignment blockers are done). The coordinator spawns two more agents. All five slots are full.

Work proceeds. Agents call `wg log` to record progress, `wg artifact` to register output files, and `wg done` when finished. Each `wg done` triggers another tick. Completed tasks unblock their dependents. The coordinator spawns new agents as slots open. If an agent crashes, the next tick detects the dead PID, triages the output, and either marks the task done, injects recovery context and reopens it, or restarts it cleanly.

The graph drains. Tasks move from open through in-progress to done. Evaluation tasks score completed work. Eventually the coordinator finds no ready tasks and all tasks terminal. It logs: "All tasks complete." The daemon continues running, waiting for new tasks. The operator adds more work with `wg add`, the graph_changed signal fires, and the cycle begins again.

This is coordination: a loop that converts a plan into action, one tick at a time.

6 Evolution & Improvement

The agency does not merely execute work. It learns from it.

Every completed task generates a signal—a scored evaluation measuring how well the agent performed against the task's requirements and the agent's own declared standards. These signals accumulate into performance records on agents, roles, and motivations. When enough data exists, an evolution cycle reads the aggregate picture and proposes structural changes: sharpen a role's description, tighten a motivation's constraints, combine two high-performers into something new, retire what consistently underperforms. The changed entities receive new content-hash IDs, linked to their parents by lineage metadata. Better identities produce better work. Better work produces sharper evaluations. The loop closes.

This is the autopoietic core of the agency system—a structured feedback loop where work produces the data that drives its own improvement.

6.1 Evaluation

Evaluation is the act of scoring a completed task. It answers a concrete question: given what this agent was asked to do and the identity it was given (see Section 4), how well did it perform?

The evaluator is itself an LLM agent. It receives the full context of the work: the task definition (title, description, deliverables), the agent's identity (role and motivation), any artifacts the agent produced, log entries from execution, and timing data (when the task started and finished). From this, it scores four dimensions:

Dimension	Weight	What it measures
Correctness	40%	Does the output satisfy the task's requirements and the role's desired outcome?
Completeness	30%	Were all aspects of the task addressed? Are deliverables present?
Efficiency	15%	Was the work done without unnecessary steps, bloat, or wasted effort?
Style adherence	15%	Were project conventions followed? Were the motivation's constraints respected?

The weights are deliberate. Correctness dominates because wrong output is worse than incomplete output. Completeness follows because partial work still has value. Efficiency and style adherence matter but are secondary—a correct, complete solution with poor style is more useful than an elegant, incomplete one.

The four dimension scores are combined into a single weighted score between 0.0 and 1.0. This score is the fundamental unit of evolutionary pressure.

6.1.1 Three-Level Propagation

A single evaluation does not merely update one record. It propagates to three levels:

1. **The agent's performance record.** The score is appended to the agent's evaluation history. The agent's average score and task count update.
2. **The role's performance record**—with the motivation's ID recorded as `context_id`. This means the role's record knows not just its average score, but *which motivation it was paired with* for each evaluation.

3. **The motivation's performance record**—with the role's ID recorded as `context_id`. Symmetrically, the motivation knows which role it was paired with.

This three-level, cross-referenced propagation creates the data structure that makes synergy analysis possible. A role's aggregate score tells you how it performs *in general*. The context IDs tell you how it performs *with specific motivations*. The distinction matters: a role might score 0.9 with one motivation and 0.5 with another. The aggregate alone would hide this.

6.1.2 What Gets Evaluated

Both done and failed tasks can be evaluated. This is intentional—there is useful signal in failure. Which agents fail on which kinds of tasks reveals mismatches between identity and work that evolution can address.

Human agents are tracked by the same evaluation machinery, but their evaluations are excluded from the evolution signal. The system does not attempt to "improve" humans. Human evaluation data exists for reporting and trend analysis, not for evolutionary pressure.

6.2 Performance Records and Aggregation

Every role, motivation, and agent maintains a performance record: a task count, a running average score, and a list of evaluation references. Each reference carries the score, the task ID, a timestamp, and the crucial `context_id`—the ID of the paired entity.

From these records, two analytical tools emerge.

6.2.1 The Synergy Matrix

The synergy matrix is a cross-reference of every (role, motivation) pair that has been evaluated together. For each pair, it shows the average score and the number of evaluations. `wg agency stats` renders this automatically.

High-synergy pairs—those scoring 0.8 or above—represent effective identity combinations worth preserving and expanding. Low-synergy pairs—0.4 or below—represent mismatches. Under-explored combinations with too few evaluations are surfaced as hypotheses: try this pairing and see what happens.

The matrix is not a static report. It is a map of the agency's combinatorial identity space, updated with every evaluation. It tells you where your agency is strong, where it is weak, and where it has not yet looked.

6.2.2 Trend Indicators

`wg agency stats` also computes directional trends. It splits each entity's recent evaluations into first and second halves and compares the averages. If the second half scores more than 0.03 higher, the trend is *improving*. More than 0.03 lower, *declining*. Within 0.03, *flat*.

Trends answer the question that aggregate scores cannot: is this entity getting better or worse over time? A role with a middling 0.65 average but an improving trend is a better evolution candidate than one with a static 0.70. Trends make the temporal dimension of performance visible.

6.3 Evolution

Evolution is the process of improving agency entities based on accumulated evaluation data. Where evaluation extracts signal from individual tasks, evolution acts on the aggregate—reading the full performance picture and proposing structural changes to roles and motivations.

Evolution is triggered manually by running `wg evolve`. This is a deliberate design choice. The system accumulates evaluation data automatically (via the coordinator's auto-evaluate feature, see Section 5.7), but the decision to act on that data belongs to the human. Evolution is powerful enough to reshape the agency's identity space. It should not run unattended.

6.3.1 The Evolver Agent

The evolver is itself an LLM agent. It receives a comprehensive performance summary: every role and motivation with their scores, dimension breakdowns, generation numbers, lineage, and the synergy matrix. It also receives strategy-specific guidance documents from `.workgraph/agency/evolver-skills/`—prose procedures for each type of evolutionary operation.

The evolver can have its own agency identity—a role and motivation that shape how it approaches improvement. A cautious evolver motivation that rejects aggressive changes will produce different proposals than an experimental one. The evolver's identity is configured in `config.toml` and injected into its prompt, just like any other agent.

6.3.2 Strategies

Six strategies define the space of evolutionary operations:

Mutation. The most common operation. Take an existing role or motivation and modify it to address specific weaknesses. If a role scores poorly on completeness, the evolver might sharpen its desired outcome or add a skill reference that emphasizes thoroughness. The mutated entity receives a new content-hash ID—it is a new entity, linked to its parent by lineage. These mutated entities are what it means for agents to "evolve"—they are new versions of roles or motivations, reshaped by performance data, that produce new agents when paired.

Crossover. Combine traits from two high-performing entities into a new one. If two roles each excel on different dimensions, crossover attempts to produce a child that inherits the strengths of both. The new entity records both parents in its lineage.

Gap analysis. Create entirely new roles or motivations for capabilities the agency lacks. If tasks requiring a skill no agent possesses consistently fail or go unmatched, gap analysis proposes a new role to fill that space.

Retirement. Remove consistently poor-performing entities. This is pruning—clearing out identities that evaluation has shown to be ineffective. Retired entities are not deleted; they are renamed to `.yaml.retired` and preserved for audit.

Motivation tuning. Adjust the trade-offs on an existing motivation. Tighten a constraint that evaluations show is being violated. Relax one that is unnecessarily restrictive. This is a targeted form of mutation specific to the motivation's acceptable and unacceptable trade-off lists.

All. Use every strategy as appropriate. The evolver reads the full performance picture and proposes whatever mix of operations it deems most impactful. This is the default.

Each strategy can be selected individually via `wg evolve --strategy mutation` or combined as the default `all`. Strategy-specific guidance documents in the evolver-skills directory give the evolver detailed procedures for each approach.

6.3.3 Mechanics

When `wg evolve` runs, the following sequence executes:

1. All roles, motivations, and evaluations are loaded. Human-agent evaluations are filtered out —they would pollute the signal, since human performance does not reflect the effectiveness of a role-motivation prompt.
2. A performance summary is built: role-by-role and motivation-by-motivation scores, dimension averages, generation numbers, lineage, and the synergy matrix.
3. The evolver prompt is assembled: system instructions, the evolver's own identity (if configured), meta-agent assignments (so the evolver knows which entities serve coordination roles), the chosen strategy, budget constraints, retention heuristics (a prose policy from configuration), the performance summary, and strategy-specific skill documents.
4. The evolver agent runs and returns structured JSON: a list of operations (create, modify, or retire) with full entity definitions and rationales.
5. Operations are applied sequentially. Budget limits are enforced—if the evolver proposes more operations than the budget allows, only the first N are applied. After each operation, the local state is reloaded so subsequent operations can reference newly created entities.
6. A run report is saved to `.workgraph/agency/evolution_runs/` with the full transcript: what was proposed, what was applied, and why.

6.3.4 How Modified Entities Are Born

When the evolver proposes a `modify_role` operation, the system does not edit the existing role in place. It creates a *new* role with the modified fields, computes a fresh content-hash ID from the new content, and writes it as a new YAML file. The original role remains untouched.

The new role's lineage records its parent: the ID of the role it was derived from, a generation number one higher than the parent's, the evolver run ID as the creator, and a timestamp. For crossover operations, the lineage records multiple parents and takes the highest generation among them.

This is where content-hash IDs and immutability (see Section 4.4) pay off. The original entity is a mathematical fact—its hash proves it has not been tampered with. The child is a new fact, with a provable link to its origin. You can walk the lineage chain from any entity back to its manually-created ancestor at generation zero.

6.4 Safety Guardrails

Evolution is powerful. The guardrails are proportional.

The last remaining role or motivation cannot be retired. The agency must always have at least one of each. This prevents an overzealous evolver from pruning the agency into nonexistence.

Retired entities are preserved, not deleted. The `.yaml.retired` suffix removes them from active duty but keeps them on disk for audit, rollback, or lineage inspection.

Dry run. `wg evolve --dry-run` renders the full evolver prompt and shows it without executing. You see exactly what the evolver would see. This is the first thing to run when experimenting with evolution.

Budget limits. `--budget N` caps the number of operations applied per run. Start small—two or three operations—review the results, iterate. The evolver may propose ten changes, but you decide how many land.

Self-mutation deferral. The evolver's own role and motivation are valid mutation targets—the system should be able to improve its own improvement mechanism. But self-modification without oversight is dangerous. When the evolver proposes a change to its own identity, the operation is not applied directly. Instead, a review meta-task is created in the workgraph with a `verify` field requiring human approval. The proposed operation is embedded in the task description as JSON. A human must inspect the change and apply it manually.

6.5 Lineage

Every role, motivation, and agent tracks its evolutionary history through a lineage record: parent IDs, generation number, creator identity, and timestamp.

Generation zero entities are the seeds—created by humans via `wg role add`, `wg motivation add`, or `wg agency init`. They have no parents. Their `created_by` field reads `"human"`.

Generation one entities are the first children of evolution. A mutation from a generation-zero role produces a generation-one role with a single parent. A crossover of two generation-zero roles produces a generation-one role with two parents. Each subsequent evolution increments from the highest parent's generation.

The `created_by` field on evolved entities records the evolver run ID: `"evolver-run-20260115-143022"`. Combined with the run reports saved in `evolution_runs/`, this creates a complete audit trail: you can trace any entity to the exact evolution run that created it, see what performance data the evolver was working from, and read the rationale for the change.

Lineage commands—`wg role lineage`, `wg motivation lineage`, `wg agent lineage`—walk the chain. Agent lineage is the most interesting: it shows not just the agent's own history but the lineage of its constituent role and motivation, revealing the full evolutionary tree that converged to produce that particular identity.

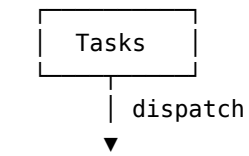
6.6 The Autopoietic Loop

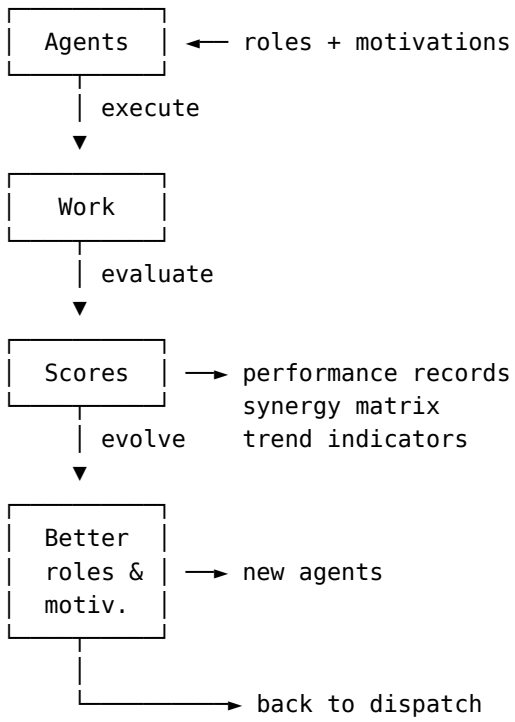
Step back from the mechanics and see the shape of the whole.

Work enters the system as tasks. The coordinator (see Section 5) dispatches agents—each carrying an identity composed of a role and a motivation (see Section 4)—to execute those tasks. When a task completes, auto-evaluate creates an evaluation meta-task. The evaluator agent scores the work across four dimensions. Scores propagate to the agent, the role, and the motivation. Over time, performance records accumulate. Trends emerge. The synergy matrix fills in.

When the human decides enough signal has accumulated, `wg evolve` runs. The evolver reads the full performance picture and proposes changes. A role that consistently scores low on efficiency gets its description sharpened to emphasize economy. A motivation whose constraints are too tight gets its trade-offs relaxed. Two high-performing roles get crossed to produce a child that inherits both strengths. A consistently poor performer gets retired.

The changed entities—new roles, new motivations—are paired into new agents. These agents are dispatched to the next round of tasks. Their work is evaluated. Their evaluations feed the next evolution cycle.





The meta-agents—the assigner that picks which agent gets which task, the evaluator that scores the work, the evolver that proposes changes—are themselves agency entities with roles and motivations. They too can be evaluated. They too can be evolved. The evolver can propose improvements to the evaluator's role. It can propose improvements to *its own* role, subject to the self-mutation safety check that routes such proposals through human review.

This is what makes the system autopoietic: it does not just produce work, it produces the conditions for better work. It does not just execute, it reflects on execution and restructures itself in response. The identity space of the agency—the set of roles, motivations, and their pairings—is not static. It is a living population subject to selective pressure from the evaluation signal and evolutionary operations from the evolver.

But the human hand is always on the wheel. Evolution is a manual trigger, not an automatic process. The human decides when to evolve, reviews what the evolver proposes (especially via `--dry-run`), sets budget limits, and must personally approve any self-mutations. The system improves itself, but only with permission.

6.7 Practical Guidance

When to evolve. Wait until you have at least five to ten evaluations per role before running evolution. Fewer than that, and the evolver is working from noise rather than signal. `wg agency stats` shows evaluation counts and trends—use it to judge readiness.

Start with dry run. Always run `wg evolve --dry-run` first. Read the prompt. Understand what the evolver sees. This also serves as a diagnostic: if the performance summary looks thin, you need more evaluations before evolving.

Use budgets. `--budget 2` or `--budget 3` for early runs. Review each operation's rationale. As you build confidence in the evolver's judgment, you can increase the budget or omit it.

Targeted strategies. If you know what the problem is—roles scoring low on a specific dimension, motivations with constraints that are too strict—use a targeted strategy. `--strategy`

`mutation` for improving existing entities. `--strategy motivation-tuning` for adjusting trade-offs. `--strategy gap-analysis` when tasks are going unmatched.

Seed, then evolve. `wg agency init` creates four starter roles and four starter motivations. These are generic seeds—competent but not specialized. Run them through a few task cycles, accumulate evaluations, then run `wg evolve`. The starters are generation zero. Evolution produces generation one, two, and beyond—each generation shaped by the actual work your project requires.

Watch the synergy matrix. The matrix reveals which role-motivation pairings work well together and which do not. High-synergy pairs should be preserved. Low-synergy pairs are candidates for mutation or retirement. Under-explored combinations are experiments waiting to happen—assign them to tasks and see what the evaluations say.

Lineage as audit. When an agent produces unexpectedly good or bad work, trace its lineage. Which evolution run created its role? What performance data informed that mutation? The lineage chain, combined with evolution run reports, makes every identity decision traceable.