Workgraph

A Manual

Task coordination for humans and AI agents

Contents

1 Glossary 1
2 System Overview 6
  2.1 The Graph Is the Work 6
  2.2 The Agency Is Who Does It 7
  2.3 The Core Loop 8
  2.4 The Agency Loop 8
  2.5 How They Relate 9
3 The Task Graph 11
  3.1 Tasks as Nodes 11
  3.2 Status and Lifecycle 13
  3.3 Terminal Statuses Unblock: A Design Choice 14
  3.4 Dependencies: `after` and `before` 14
  3.5 Readiness 15
  3.6 Structural Cycles: Intentional Iteration 15
  3.7 Pause and Resume 19
  3.8 Placement Hints 19
  3.9 Emergent Patterns 19
  3.10 Graph Analysis 21
  3.11 Storage 22
4. The Agency Model 24
  4.1. Roles 24
  4.2. Motivations 24
  4.3. Agents: The Pairing 25
  4.4. Content-Hash IDs 26
  4.5. The Skill System 26
  4.6. Trust Levels 27
  4.7. Human and AI Agents 27
  4.8. Composition in Practice 28
  4.9. Lineage and Deduplication 29
  4.10. Federation: Sharing Across Projects 29
  4.11. Automation: Auto-Create and Auto-Place 30
  4.12. Configuration: Creator Identity 30
  4.13. Cross-References 30
5 Coordination & Execution 32
  5.1 The Service Daemon 32
  5.2 The Coordinator Tick 32
  5.3 The Dispatch Cycle 34
  5.4 The Wrapper Script 36
  5.5 Parallelism Control 37
  5.6 Auto-Assign 37
  5.7 Auto-Evaluate 38
  5.8 Dead Agent Detection and Triage 38
  5.9 IPC Protocol 39
  5.10 Multi-Coordinator Sessions 40
  5.11 Compaction, Sweep, and Checkpoint 40
  5.12 Service Restart 40

5.13 Observing the System ... 40
5.14 Custom Executors ... 42
5.15 Pause, Resume, and Manual Control ... 42
5.16 The Full Picture ... 42
6 Evolution & Improvement ... 44
6.1 Evaluation ... 44
6.2 Performance Records and Aggregation ... 46
6.3 Evolution ... 47
6.4 Safety Guardrails ... 49
6.5 Lineage ... 49
6.6 The Autopoietic Loop ... 50
6.7 Practical Guidance ... 52

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 eight values: *open* (available for work), *in-progress* (claimed by an agent), *done* (completed successfully), *failed* (attempted and failed; retryable), *abandoned* (permanently dropped), *blocked* (explicit, rarely used), *pending-validation* (awaiting review after `wg done` on a task with `verify` criteria), or *waiting* (parked by `wg wait` until a condition is met). The three *terminal* statuses are done, failed, and abandoned—a terminal task no longer blocks its dependents. PendingValidation and Waiting are non-terminal intermediate states.
dependency	A directed edge between tasks expressed via the `after` field. Task B depends on task A means B cannot be ready until A reaches a terminal status.
after	The authoritative dependency list on a task. `task.after = ["dep"]` means the task comes after `dep`. A task is *waiting* (in the derived sense) when any entry in its `after` list is non-terminal. In the CLI, specified via `--after` (alias: `--blocked-by`).
before	The computed inverse of `after`, maintained for bidirectional traversal. If B is after A, then A's `before` list includes B. 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 `after` list is terminal. For cycle headers, back-edge predecessors are exempt.
structural cycle	A cycle formed by `after` edges, detected automatically by Tarjan's SCC algorithm. Each cycle has a header (entry point) with a `CycleConfig` controlling iteration. Replaces the former `loops_to` edge system.
CycleConfig	Configuration for cycle iteration, stored on the cycle header task. Fields: `max_iterations` (hard cap), `guard` (optional condition), `delay` (optional pacing between iterations).
guard	A condition on a cycle's `CycleConfig`. Three kinds: *Always*, *TaskStatus*, and *IterationLessThan*.
loop iteration	A counter tracking how many times a task has been re-activated by cycle iteration.

Term	Definition
visibility	A field on each task controlling what information crosses organizational boundaries during trace exports. Three values: *internal* (default, org-only), *public* (sanitized sharing—task structure without agent output or logs), *peer* (richer detail for trusted peers—includes evaluations but strips notes and detailed logs). Set via `wg add --visibility` or `wg edit`.
convergence	An agent-driven signal (`wg done --converged`) indicating that a cycle's iterative work has reached a stable state. Adds a `"converged"` tag to the cycle header (regardless of which member the agent completes). When the header carries this tag, the cycle does not iterate—even if iterations remain and guards are satisfied. Cleared on retry.
trace	The operations log (`operations.jsonl`) recording every mutation to the graph. The project's organizational memory—queryable via `wg trace`, exportable with visibility filtering, and importable from peers.
trace export	A filtered, shareable snapshot of the trace. Visibility filtering controls what is included: *internal* exports everything, *public* sanitizes, *peer* provides richer detail for trusted peers. Produced by `wg trace export --visibility <zone>`.
function	A parameterized workflow template extracted from completed traces via `wg func extract`. Captures task structure, dependencies, and structural cycles. Applied via `wg func apply` to create new task graphs. Stored as YAML in `.workgraph/functions/`.
replay	Re-execution of previously completed or failed work. `wg replay` creates an immutable snapshot, then selectively resets tasks based on criteria. Supports `--plan-only` for previewing.
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 (tradeoff)	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. Called *tradeoff* in the CLI (`wg tradeoff`); the older name *motivation* is accepted as an alias.
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/`).

Term	Definition
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.
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.

Term	Definition
evaluation source	A freeform string tag on each evaluation identifying its origin. Default: `"llm"` (internal auto-evaluator). Conventions: `"outcome:<metric>"` for external outcome data, `"ci:<suite>"` for CI results, `"vx:<peer-id>"` for peer evaluations. The evolver reads all evaluations regardless of source.
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*, *tradeoff 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 completes, enabling parallel children) and fan-in (parallel tasks must all complete before 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.
federation	The system for sharing agency entities across workgraph projects. Operations: *scan* (discover), *pull* (import), *push* (export). Named remotes stored in `.workgraph/federation.yaml`. Content-hash IDs make deduplication automatic.
remote	A named reference to another workgraph project's agency store, used for federation. Managed via `wg agency remote add/list/remove`.

Term	Definition
provider	The LLM provider backing a task or agent: `anthropic`, `openai`, `openrouter`, or `local`. Set per-task via `--provider` on `wg add`/ `wg edit`, or per-agent via `wg config`. The coordinator resolves providers through the same priority chain as models.
exec-mode	Controls the execution weight of an agent dispatched for a task. Four values: *full* (default—complete tool access), *light* (read-only tools), *bare* (only `wg` CLI), *shell* (no LLM—runs the task's `exec` field directly). Set via `--exec-mode` on `wg add`/`wg edit`.
placement	The coordinator's automatic positioning of newly created tasks in the dependency graph. Controlled by placement hints: `--no-place` (skip placement—make the task immediately available), `--place-near <IDS>` (place near specified tasks), `--place-before <IDS>` (insert before specified tasks). Automatic placement is configured via `wg config --auto-place`.
multi-coordinator	Support for running multiple coordinator sessions within a single service daemon. Each coordinator manages an independent scheduling context. Managed via `wg service create-coordinator`, `stop-coordinator`, `archive-coordinator`, and `delete-coordinator`. The maximum number of concurrent coordinators is set via `wg config --max-coordinators`.
compaction	The process of distilling graph state into a condensed summary (`context.md`). Triggered via `wg compact`. In the service daemon, compaction runs as the `.compact-0` task—a structural cycle where the coordinator introspects its own state.
sweep	Detection and recovery of orphaned in-progress tasks whose agents have died. `wg sweep` scans for tasks claimed by agents whose PIDs no longer exist and offers to reclaim them.
checkpoint	A snapshot of an agent's progress during long-running tasks. `wg checkpoint` saves the current state so that if the agent is interrupted, a replacement can resume from the checkpoint rather than starting over.
event stream	A real-time feed of graph mutations produced by `wg watch`. Events are typed (`task.created`, `task.completed`, `evaluation.recorded`, etc.) and filterable by category or task ID. Enables external adapters to observe and react without polling.
adapter	An external tool that translates between an external system's vocabulary and workgraph's ingestion points. The generic pattern: observe (via `wg watch`) → translate → ingest (via `wg` CLI) → react. A conceptual pattern, not a formal type.

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 structural cycles introduce intentional iteration 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 2*, the agency model in *Section 3*, coordination and execution in *Section 4*, and evolution in *Section 5*.

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 `after` field. If task B lists task A in its `after` list, then B comes after A—it 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. *Structural cycles*—cycles that form naturally in the `after` edges—enable intentional iteration. A write-review-revise cycle, a CI retry pipeline, a monitoring loop: all are expressible as dependency chains where an `after` edge points backward, creating a cycle detected automatically by the system. The cycle's header task carries a `CycleConfig` with a mandatory `max_iterations` cap and an optional *guard* condition. The header receives a back-edge exemption in the readiness check, allowing the cycle to start and iterate. When all cycle members complete and the guard is satisfied, the entire cycle re-opens for the next iteration. When iterative work reaches a stable state before exhausting its iteration budget, any agent in the cycle can signal convergence to halt early.

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 *tradeoff* (also called a *motivation* in prose) defines *why* the agent acts the way it does: its priorities, its acceptable trade-offs, and its hard constraints. The CLI command is `wg tradeoff`; this manual uses "motivation" when discussing the concept and "tradeoff" when referencing commands. 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 amount of surrounding context included in the prompt is controlled by a *context scope*: `clean` (bare executor, no workflow instructions), `task` (standard default with workflow commands and graph patterns), `graph` (adds project description and 1-hop neighborhood summary), or `full` (adds complete graph summary, CLAUDE.md, and system preamble). Each tier is a strict superset of the one below. The scope is resolved from a priority chain: task-level override, then role default, then coordinator configuration, then the default of `task`. The agent doesn't just receive a task description; it receives an identity and a calibrated view of the project. This is what separates a workgraph agent from a one-off LLM call.

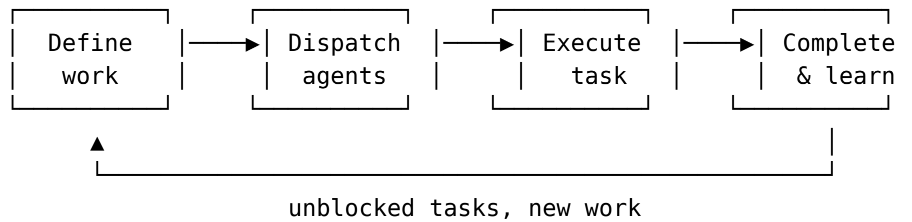
Human agents participate in the same model. The only difference is the *executor*: AI agents use `claude` (or another LLM backend); human agents use `matrix`, `email`, `shell`, or another human-facing channel. Human agents don't need roles or motivations—they bring their own judgment. But both human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

AI tasks can also specify a *provider*—`anthropic`, `openai`, `openrouter`, or `local`—and an *exec-mode* that controls the agent's level of autonomy: `full` (complete tool access), `light` (read-only), `bare` (CLI only), or `shell` (no LLM). These per-task controls let you match the execution environment to the work: a sensitive review task might use a different provider and a read-only exec-mode, while an implementation task uses the default.

Because identities are content-hashed, they travel well. Agency entities—roles, motivations, and their evaluation histories—can be shared across projects through federation, carrying lineage and performance data intact. A proven architect role in one project can be pulled into another without re-creation; the content-hash guarantees it is the same entity everywhere.

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.

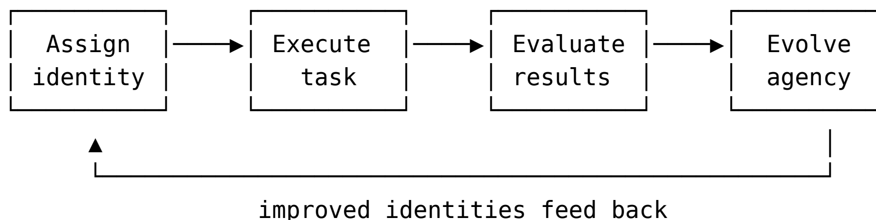
Execute. The spawned agent does its work. It may log progress, record artifacts, create subtasks, or mark the task done or failed. It operates with full autonomy within the boundaries set by its role and motivation.

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. 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. A single service daemon can host multiple coordinator sessions, enabling parallel workstreams within the same project. The graph is the skeleton; the agency is the musculature; the coordinator is the nervous system.

Several additional mechanisms extend this core architecture:

- **Waiting and checkpointing.** An agent can park a task in *waiting* status (`wg wait`) until a condition is met—another task reaching a state, a timer expiring, a message arriving, or a human signal. The coordinator evaluates waiting conditions each tick and resumes satisfied tasks automatically. Separately, `wg checkpoint` lets a running agent save a progress snapshot so that a replacement agent can resume from that point if the original is interrupted.
- **FLIP (Fidelity via Latent Intent Probing).** After a task completes and is evaluated, an independent FLIP assessment reconstructs what the task's prompt *must have been* from the agent's output alone, then scores how well the output matched the actual task description. Low FLIP scores automatically trigger verification tasks dispatched to a stronger model. The full agency pipeline is: evaluate → FLIP → verify → evolve.
- **Eval gate.** A configurable threshold (`eval_gate_threshold`) that automatically rejects (fails) a completed task if its evaluation score falls below the minimum. This creates a quality floor: work that does not meet the bar is sent back rather than accepted.
- **Multi-coordinator sessions.** A single daemon can host multiple coordinator sessions, each managing an independent scheduling context—for example, one coordinator for feature work and another for maintenance.

- **Compaction and sweep.** Long-running projects accumulate state. `wg compact` distills the current graph into a condensed `context.md` summary for future agent prompts. `wg sweep` detects orphaned in-progress tasks whose agents have died without cleanup.
- **Auto-triage.** When a spawned agent dies, the coordinator can automatically triage the outcome using a fast LLM, classifying the result as *done* (work was complete), *continue* (inject recovery context and re-dispatch), or *restart* (begin fresh).
- **Exec-mode.** Each task can specify an execution weight—`full` (all tools), `light` (read-only), `bare` (CLI only), or `shell` (no LLM)—controlling how much autonomy the agent has within its executor.

Workgraph is not a closed system. External tools—CI pipelines, portfolio trackers, peer organizations—can observe the graph through a real-time event stream and inject information back through several channels: recording evaluations with external source tags, importing trace data from peers, adding tasks, or updating state directly. Each task carries a *visibility* field (`internal`, `public`, or `peer`) that controls what information crosses organizational boundaries when traces are exported. This boundary discipline makes collaboration possible without exposing internal deliberation.

Everything is files. The graph is JSONL. Agency entities—roles, motivations, agents—are YAML. Configuration is TOML. Evaluations are YAML. Underneath it all, an operations log records every mutation to the graph—the project's trace. This trace is organizational memory: queryable for provenance, exportable for cross-boundary sharing with visibility filtering, and extractable into parameterized workflow templates that capture proven patterns for reuse. 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 eight 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.
`provider`	LLM provider for this task (`anthropic`, `openai`, `openrouter`, `local`). Overrides coordinator and agent defaults.
`exec_mode`	Execution weight controlling the agent's tool access. One of `full` (default—complete tool access), `light` (read-only tools), `bare` (only `wg` CLI), or `shell` (no LLM—runs the `exec` field directly).
`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.
`visibility`	Controls what information crosses organizational boundaries during trace exports. One of `internal` (default—organization only), `public` (sanitized sharing without agent output or logs), or `peer` (richer detail for trusted peers, including evaluations and patterns).
`context_scope`	Controls how much context the agent receives in its prompt. One of `clean` (bare executor), `task` (default—workflow commands and graph patterns), `graph` (adds project description and 1-hop neighborhood), or `full` (adds complete graph summary and CLAUDE.md). Each tier is a strict superset of the one below. Overrides role and coordinator defaults when set.
`delay`	Duration to wait before the task becomes ready (e.g., `30s`, `5m`, `1h`, `1d`). Set via `--delay` on `wg add`/`wg edit`.
`not_before`	Absolute ISO 8601 timestamp before which the task will not be dispatched. Set via `--not-before` on `wg add`/`wg edit`.
`log`	Append-only progress entries with timestamps and optional actor attribution.

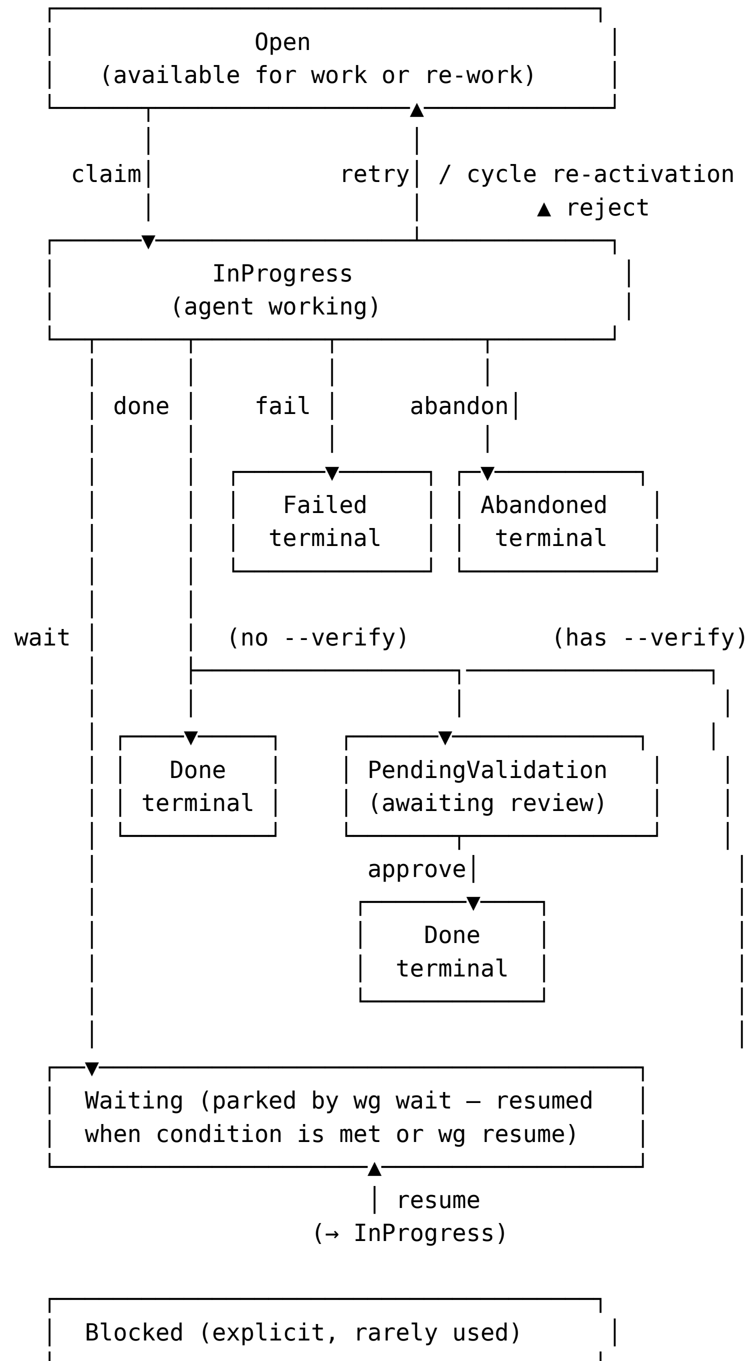
Table 1: Task fields. Every field except `id`, `title`, and `status` is optional.

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 eight statuses. Most follow the happy path; some take detours.



Listing 3: Task state machine. The eight statuses include three terminals (Done, Failed, Abandoned) that unblock dependents, plus PendingValidation and Waiting as non-terminal intermediate states.

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 cycle 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.

PendingValidation is entered when an agent calls `wg done` on a task that has `verify` criteria. The task is not yet terminal—it awaits external review. `wg approve` transitions it to Done; `wg reject` transitions it back to Open for re-dispatch (with the assignment cleared). After `max_rejections` (default: 3), rejection transitions the task to Failed instead.

Waiting means an agent has voluntarily parked the task via `wg wait`. The task is not terminal and will not be re-dispatched. It resumes (returning to InProgress) when its wait condition is met—a timer elapses, a dependent task reaches a target status, a message arrives, or a file changes—or when a human runs `wg resume`.

Blocked exists as an explicit status but is rarely used. In practice, a task is *blocked* when its `after` list contains non-terminal entries—this is a derived condition, not a declared status. 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.

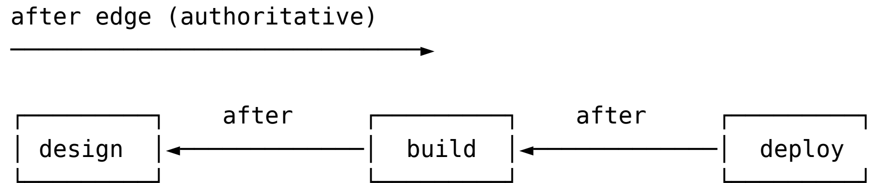
When task A fails and task B depends on A, B becomes ready. B's agent receives context from A—the failure reason, the log entries, the artifacts (if any). The agent can then decide: retry the work itself, produce a partial result, or fail explicitly with its own reason. The graph keeps moving.

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: `after` and `before`

Dependencies are directed edges expressing temporal ordering. 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 `after` list—B comes *after* A.




```
Read as: build is after design. deploy is after build.
Equivalently: design is before build. build is before deploy.
```

Listing 4: Dependency edges. `after` is authoritative; `before` is its computed inverse.

The `after` list is the source of truth. The `before` list is its inverse, maintained for bidirectional traversal—if B is after A, then A's `before` list includes B. The scheduler never reads `before`; it only checks `after`. 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 after B and B is after 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 predecessors, and the chain resolves itself one link at a time.

A subtlety: if a task references a predecessor 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 cycle delays), the current time must be past that too. Invalid or missing timestamps are treated as satisfied—they do not prevent readiness.
4. **All predecessors terminal.** Every task ID in the `after` list must correspond to a task in a terminal status (done, failed, or abandoned). Non-existent predecessors 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 cycle delays, creating pacing between cycle iterations. Both are checked against the current wall-clock time.

3.6 Structural Cycles: Intentional Iteration

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 *structural cycles*—cycles that emerge naturally from `after` edges in the task graph, detected automatically by the system.

3.6.1 How Structural Cycles Work

A structural cycle is a set of tasks whose `after` edges form a cycle. If task A is after task C, task C is after task B, and task B is after task A, the system detects this cycle automatically using Tarjan's SCC (strongly connected component) algorithm. No special edge type is needed —the cycle is a structural property of the graph.

Each cycle has a *header*: the entry point, identified as the task with predecessors outside the cycle. The header carries a `CycleConfig` that controls iteration:

Field	Purpose
`max_iterations`	Hard cap on how many times the cycle can iterate. Mandatory—no unbounded cycles.
`guard`	A condition that must be true for the cycle to iterate. Optional—if absent, the cycle iterates unconditionally (up to `max_iterations`).
`delay`	Optional duration (e.g., `"30s"`, `"5m"`, `"1h"`) to wait before the next iteration. Sets the header's `ready_after` timestamp.
`no_converge`	When set, agents cannot signal early convergence via `--converged`. All iterations (up to `max_iterations`) are forced to run. Set via `--no-converge` on `wg add`/`wg edit`.
`restart_on_failure`	Whether to automatically restart the cycle when a member fails (default: true). Disabled via `--no-restart-on-failure`. The `max_failure_restarts` field caps the number of failure-triggered restarts (default: 3).

Table 2: CycleConfig fields on the cycle header task. Every configured cycle requires a `max_iterations` cap.

The critical insight: the cycle header receives a *back-edge exemption* in the readiness check. Normally, a task is waiting when any of its `after` predecessors is non-terminal. But the header's predecessors within the cycle (the back-edges) are exempt—this allows the header to become ready on the first iteration even though its cycle predecessors have not yet completed. Non-header tasks in the cycle still wait for their predecessors normally, so the cycle executes in order from the header through the body.

A cycle without a `CycleConfig` on any member is flagged by `wg check` as an unconfigured deadlock—it will not iterate and the header will not receive back-edge exemption.

3.6.2 Guards

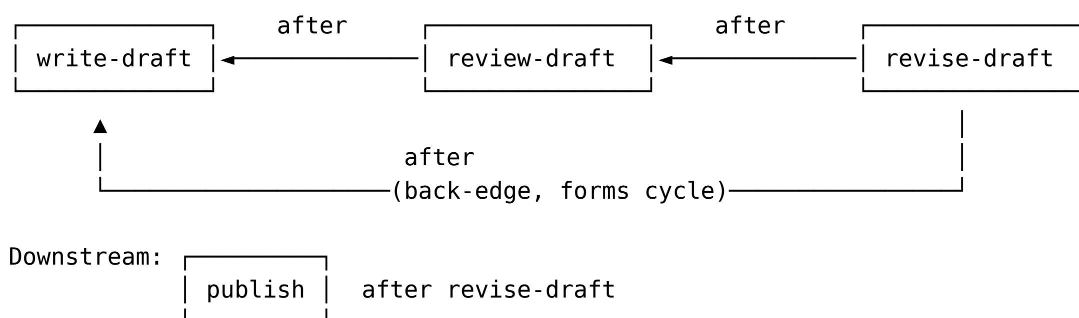
A guard is a condition on a cycle's `CycleConfig` that controls whether the cycle iterates. Three kinds:

- **Always.** The cycle iterates unconditionally on every completion, up to `max_iterations`. Used for monitoring loops and fixed-iteration patterns.
- **TaskStatus.** The cycle iterates only if a named task has a specific status. The classic use: "iterate back to writing if the review task failed." This is the mechanism for conditional retry.
- **IterationLessThan.** The cycle iterates only if the header'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 cycle behaves as `Always`—it iterates on every completion up to the iteration cap.

3.6.3 A Review Cycle, Step by Step

Consider a three-task review cycle:




```
write-draft has CycleConfig: max_iterations=5,
guard=task:review-draft=failed
```

Listing 5: A structural cycle. All edges are `after` edges. The back-edge from `write-draft` to `revise-draft` creates the cycle.

The cycle is detected automatically: `write-draft` → `review-draft` → `revise-draft` → `write-draft`. The header is `write-draft` (it has external predecessors or is the entry point). Its `CycleConfig` sets `max_iterations: 5` and a guard condition.

Created with:

```
wg add "write-draft" --max-iterations 5 --cycle-guard "task:review-draft=failed"
wg add "review-draft" --after write-draft
wg add "revise-draft" --after review-draft
wg add "publish" --after revise-draft
```

Then create the back-edge that forms the cycle:

```
wg edit write-draft --add-after revise-draft
```

Here is the execution:

1. `write-draft` is the cycle header. Its back-edge predecessor (`revise-draft`) is exempt from the readiness check. 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 predecessors 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 predecessors 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. All cycle members are now terminal. The system evaluates cycle iteration: the guard checks `review-draft`'s status—it is `Failed`. The header's `loop_iteration` is 0, below `max_iterations` (5). The cycle iterates.
8. All cycle members are re-opened: status set to `Open`, assignments and timestamps cleared, `loop_iteration` incremented to 1. A log entry records: "Re-activated by cycle iteration (iteration 1/5)."
9. `write-draft` is again ready (back-edge exemption). The cycle begins again.

If the reviewer eventually approves (calls `wg done review-draft` instead of `wg fail`), then when all members complete, the guard checks `review-draft`'s status—it is `Done`, not `Failed`. The guard condition is not met. The cycle does not iterate. All members stay done. `publish` has all predecessors terminal. The graph proceeds.

3.6.4 Cycle Re-Opening

When a cycle iterates, *all* cycle members are re-opened simultaneously. The system knows exactly which tasks belong to the cycle through SCC analysis—no BFS traversal needed. Every member's status is set to `Open`, its assignment and timestamps are cleared, and its `loop_iteration` is incremented to match the new iteration count.

This ensures the entire cycle is available for re-execution, and every member's status accurately reflects the cycle state.

3.6.5 Bounded Iteration

Every cycle header must specify `max_iterations` in its `CycleConfig`. There are no unbounded cycles. When the header's `loop_iteration` reaches the cap, the cycle stops iterating, regardless of guard conditions. All members stay done. Downstream work proceeds.

This is a safety property. A guard condition with a logic error could iterate indefinitely; `max_iterations` guarantees that every cycle terminates.

3.6.6 Early Convergence

The iteration cap is a ceiling, not a target. In practice, iterative work often converges before the maximum is reached—a refine agent determines the output is stable, a review cycle approves on the third pass instead of the fifth, a monitoring check finds the system healthy. Running all remaining iterations after convergence wastes compute and delays downstream work.

Any agent working on a cycle member can signal convergence by running `wg done <task-id> --converged`. This marks the task as done and adds a `"converged"` tag to the *cycle header* (regardless of which member the agent completes). When the cycle evaluator checks whether to iterate, it sees the tag on the header and stops—the cycle does not iterate, regardless of guard conditions or remaining iterations. Downstream tasks proceed immediately.

The convergence tag is durable but not permanent. Running `wg retry` on a converged task clears the tag along with resetting the task to open, so the cycle can iterate again if needed. This means convergence is an agent's assertion about *this* iteration's outcome, not a permanent lock on the cycle structure.

The coordinator supports this mechanism in the dispatch cycle: when rendering a prompt for a task that is part of a structural cycle, it includes a note about the `--converged` flag, informing the agent that early termination is available. The agent decides—the system does not guess.

3.6.7 Cycle Delays

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

This creates pacing between iterations. A monitoring cycle that checks system health every five minutes uses a delay of `"5m"`. A review cycle 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 cycle—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 cycle 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-cycle to halt iteration without losing state. When you resume, the cycle picks up where it left off.

3.8 Placement Hints

When a new task is added, the coordinator can automatically position it in the dependency graph through *placement*—an optional feature controlled by `wg config --auto-place`. Placement hints on `wg add` guide this positioning:

- `--no-place` skips automatic placement entirely, leaving the task with only the dependencies explicitly specified via `--after`.
- `--place-near <IDS>` suggests placing the task near the specified tasks in the graph—useful for grouping related work.
- `--place-before <IDS>` suggests inserting the task before the specified tasks, adding dependency edges so those tasks come after the new one.

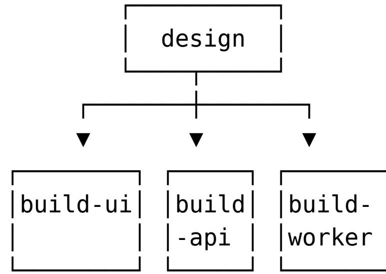
Placement is a convenience, not a constraint. All dependency edges it creates are ordinary `after` edges, visible in the graph and editable with `wg edit`. If placement produces an undesirable result, adjust the edges manually.

3.9 Emergent Patterns

The dependency edges (`after`/`before`) and structural cycles are the only primitives. But from these mechanisms, several structural patterns emerge naturally.

3.9.1 Fan-Out (Map)

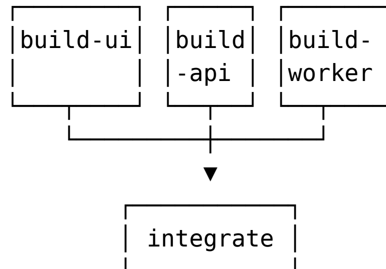
One task is before several children. When the parent completes, all children become ready simultaneously and can execute in parallel.



Listing 6: Fan-out: one parent completes, enabling parallel children.

3.9.2 Fan-In (Reduce)

Several tasks are before a single aggregator. The aggregator becomes ready only when all of its predecessors 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.

3.9.3 Pipelines

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

3.9.4 Review Cycles

A dependency chain with a back-edge creating a structural cycle, as described above. The cycle executes repeatedly until a guard condition breaks it, convergence is signaled, or the iteration cap is reached. Review cycles are the canonical example of intentional iteration.

3.9.5 Functions: Reusable Patterns

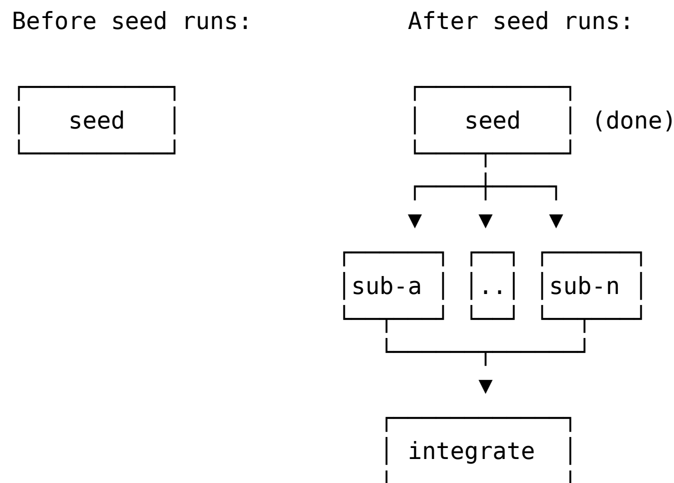
When a workflow pattern proves useful—a review cycle that consistently produces good results, a map/reduce pipeline tuned for a particular domain—it can be extracted from a completed trace into a reusable template called a *function*. The `wg func extract` command takes a completed task and its subgraph, captures the task structure, dependencies, structural cycles, and guards, and parameterizes the variable parts: feature names, file paths, descriptions, and thresholds become named input variables. The result is stored as YAML in `.workgraph/functions/`.

Applying a function with `wg func apply` reverses the process. It takes a function name and a set of input values, substitutes them into the template, and creates concrete tasks in the graph with proper dependency wiring. The original pattern's structure is preserved—its fan-out topology, its cycle bounds, its guard conditions—but applied to new work. Functions can

also be shared across projects: the `--from` flag accepts a peer name or file path, enabling teams to import proven workflows from one another.

3.9.6 Seed Tasks (Generative Tasks)

A *seed task* is a task whose primary purpose is to bootstrap a subgraph—it fans out into subtasks that did not exist before it ran. The seed does not do the "real" work itself; it analyzes a problem, decomposes it into concrete steps, and creates the tasks that perform those steps. Once the seed completes, the graph has new structure that the coordinator dispatches.



Listing 8: A seed task creates structure. The graph before execution has one node; the graph after has many.

The seed pattern is common in practice:

- A planning task that reads a spec, identifies components, and creates one implementation task per component.
- A research task that surveys a topic, identifies sub-questions, and creates investigation tasks for each.
- A triage task that reads an incoming report and creates the appropriate response tasks.

Seed tasks are often the root of a diamond (fan-out then fan-in) or scatter-gather topology. What distinguishes a seed from an ordinary fan-out parent is that the children *do not exist in the graph until the seed runs*. The graph is not just executed—it is grown.

In theoretical terms, a seed task is a *generative task*: it produces the network components that constitute the next phase of work. This connects to Maturana and Varela's concept of autopoiesis—a production relation where the system produces its own components. The seed is the autopoietic act at the task level: a node that produces nodes.

Casually, seed tasks are sometimes called *spark tasks*—the spark that ignites a subgraph into existence.

3.10 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.

Visualization. `wg viz` renders the graph as text. The `--graph` format produces a 2D spatial layout using Unicode box-drawing characters, positioning tasks by their dependency depth—roots at the top, leaf tasks at the bottom. Nodes are color-coded by status and connected by vertical lines that split at fan-out points and merge at fan-in points. The layout algorithm assigns layers via topological sort, then orders nodes within each layer to minimize edge crossings.

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

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

Listing 9: 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.

Alongside the graph file, the operations log (`operations.jsonl`) records every mutation: task creation, status changes, dependency additions, cycle iterations, evaluations. This log is the

project's trace—its organizational memory. The `wg trace` command queries it. `wg trace export` produces a filtered, shareable snapshot with visibility controls: an `internal` export includes everything, a `public` export sanitizes (omitting agent output and logs), and a `peer` export provides richer detail for trusted collaborators. `wg trace import` ingests a peer's export, enabling cross-boundary knowledge transfer. The graph file tells you where the project *is*. The operations log tells you how it got there.

—

The task graph is the foundation. Dependencies (via `after` edges) encode the ordering constraints of reality. Structural cycles 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 the section on coordination and execution. The agency system uses the graph to record evaluations at each task boundary, as described in the section on evolution.

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 (called a *tradeoff* in the CLI—`wg tradeoff`) 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 mutable operational fields that do not affect its identity: a *name* (a human-readable label like "Programmer" or "Architect"), a *performance* record (aggregated evaluation scores), *lineage* metadata (evolutionary history), an optional *context scope* default (`clean`, `task`, `graph`, or `full`), and an optional *exec-mode* default (`full`, `light`, `bare`, or `shell`). When an agent with this role is dispatched, the role's context scope and exec-mode are used as fallbacks if the task does not specify them (see the resolution priority chains in Section 5). 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

> **Terminology note:** The CLI uses the command `wg tradeoff` for this concept. This manual uses "motivation" when discussing the conceptual model and "tradeoff" when referencing CLI commands.

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, 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 (detailed in *Section 5*) 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.

4.10. Federation: Sharing Across Projects

An agency built in one project is not confined to that project. The federation system lets you share roles, motivations, and agents across workgraph projects—transferring proven identities from one context into another, complete with their performance histories and lineage chains.

Federation operates through named *remotes*: references to another project's agency store, managed via `wg agency remote add`, `wg agency remote list`, and `wg agency remote remove`. Remotes are stored in `.workgraph/federation.yaml`. Once a remote is configured, three operations become available.

Scanning. `wg agency scan <remote>` lists the roles, motivations, and agents in a remote store without modifying anything. This is reconnaissance—you see what exists before deciding what to import.

Pulling. `wg agency pull <remote>` copies entities from the remote store into the local project. Roles, motivations, agents, and their evaluation records are all transferred. You can filter by entity type (`--roles-only`, `--motivations-only`) or by specific entity IDs. A `--dry-run` flag previews the operation without writing.

Pushing. `wg agency push <remote>` is the symmetric operation—it copies local entities to the remote store. The same filtering and dry-run options apply.

Content-hash IDs make federation natural. Because identity is determined by content, the same role has the same ID in every project. When you pull a role that already exists locally, the system recognizes the collision and skips the duplicate. There is no mapping table, no namespace negotiation, no manual reconciliation. Identity deduplication is a mathematical consequence of content-hashing.

The interesting question is what happens to *metadata*—the mutable fields that sit outside the content-hash. Performance records are merged: evaluation references from both stores are unioned, deduplicated by the `(task_id, timestamp)` tuple, and average scores are recalculated from the merged set. This means pulling from a remote enriches the local performance picture—

you gain evaluation data from contexts you have never seen. Lineage is preserved by preferring the richer ancestry: if the remote's lineage records more parents or a higher generation, it takes precedence. Names default to keeping the local value, though a `--force` flag overrides this.

Referential integrity is enforced during transfer. When you pull an agent, its referenced role and motivation are automatically included—you cannot end up with an agent pointing to a role that does not exist. If a dependency is missing from the source store, the operation fails with a clear error rather than creating a broken reference.

Federation preserves lineage across project boundaries. An entity pulled from a remote carries its full ancestry chain. You can trace it back through mutations and crossovers to its manually created roots, even when those roots were created in a different project by a different team. The immutable nature of content-hash IDs guarantees that each link in the chain refers to the exact content that existed at creation time, no matter where it was created.

The practical effect is that organizations can maintain a shared pool of proven agent identities. A team that has evolved an effective "Reviewer" role over dozens of evaluations can push it to a shared remote. Other teams pull it, pair it with their own motivations, and immediately benefit from that evolutionary history. The performance data travels with the entity, so the receiving team can see *why* the role is considered effective before deciding to adopt it.

4.11. Automation: Auto-Create and Auto-Place

Two configuration options streamline the agency pipeline for projects that want minimal manual intervention:

- `auto_create` (set via `wg config --auto-create`) tells the coordinator to automatically create agent identities for new tasks based on the available roles and motivations. Without it, agents must be explicitly created and assigned.
- `auto_place` (set via `wg config --auto-place`) enables automatic placement of newly added tasks in the dependency graph. The coordinator uses heuristics to position the task near related work, respecting any placement hints (`--place-near`, `--place-before`) provided at creation time.

Both options interact with the existing `auto_assign` pipeline: when all three are enabled, a new task is automatically placed, assigned an agent identity, and dispatched—the full lifecycle from creation to execution requires no manual intervention beyond the initial `wg add`.

4.12. Configuration: Creator Identity

The agency configuration supports two settings that control the identity recorded on newly created entities: `creator_agent` and `creator_model`. These are set via `wg config --creator-agent <agent-hash>` and `wg config --creator-model <model>`. When configured, new roles, motivations, and agents created by the system record these values in their metadata, providing provenance for entities created programmatically (e.g., by the evolver or by automated workflows).

4.13. Cross-References

The agency model described here is the *identity layer* of the system. How these identities are dispatched to tasks—the claim-before-spawn protocol, the wrapper script, the coordinator's tick loop—is detailed in *Section 4*. How agents are evaluated after completing work, and how evaluation data feeds back into evolution, is detailed in *Section 5*.

One detail bridges the agency model and the evaluation system: every evaluation carries a `source` field that identifies where the score came from. Internal auto-evaluations have source `"llm"`. External signals use structured tags—`"outcome:sharpe"` for market data, `"ci:test-suite"` for CI results, `"vx:peer-id"` for peer assessments. The source field is a freeform string, not a closed enum, so any signal source can participate. This matters for the agency model because an agent's performance record aggregates evaluations from *all* sources. The evolver sees the full picture: internal quality assessments alongside external outcome data. The interplay between diverse evaluation sources and the evolutionary process is detailed in *Section 5*.

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 proceeds through a series of phases. A preliminary phase zero processes the coordinator's chat inbox (user-facing messages that arrived since the last tick). Then the numbered phases run:

Phase 1: 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.

Phase 1.3: Zero-output agent detection. Agents alive for five or more minutes with zero bytes written to their output stream are considered zombies—processes that launched but never produced work (typically due to API failures or stuck sessions). The coordinator kills these

agents and unclaims their tasks. A three-layer circuit breaker prevents cascading waste: at the *agent* level, the zombie is killed immediately; at the *per-task* level, after two consecutive zero-output spawns for the same task, the task is failed rather than retried; at the *global* level, if 50% or more of alive agents are zero-output, the coordinator pauses all spawning with exponential backoff (60 seconds up to a 15-minute maximum), preventing the system from burning compute against a downed API.

Phase 1.5: Auto-checkpoint alive agents. The coordinator saves a checkpoint for agents that have exceeded a configured turn count or elapsed time threshold. This preserves context for recovery if the agent is later killed or dies unexpectedly. A replacement agent can resume from the checkpoint rather than starting from scratch.

Phase 2: Load graph. The graph file is read from disk.

Phase 2.5: Cycle iteration evaluation. If all members of a structural cycle are done and the cycle has not converged or hit `max_iterations`, the coordinator re-opens all members for the next iteration.

Phase 2.6: Cycle failure restart. If a cycle member has failed and `restart_on_failure` is true (the default in `CycleConfig`), the coordinator re-activates the cycle for another attempt. This prevents a single transient failure from permanently halting an iterative workflow.

Phase 2.7: Wait/resume evaluation. The coordinator checks all tasks in *waiting* status for satisfied conditions—another task reaching a specified state, a timer expiring, a message arriving, or a human signal. Satisfied tasks transition back to *open*. The coordinator also detects and fails circular waits (task A waiting on task B waiting on task A).

Phase 2.8: Message-triggered resurrection. Done tasks that have unread messages from whitelisted senders (the user, the coordinator, or dependent-task agents) are reopened so the next agent can address the message. Rate-limited to a maximum of three resurrections per task with a cooldown period.

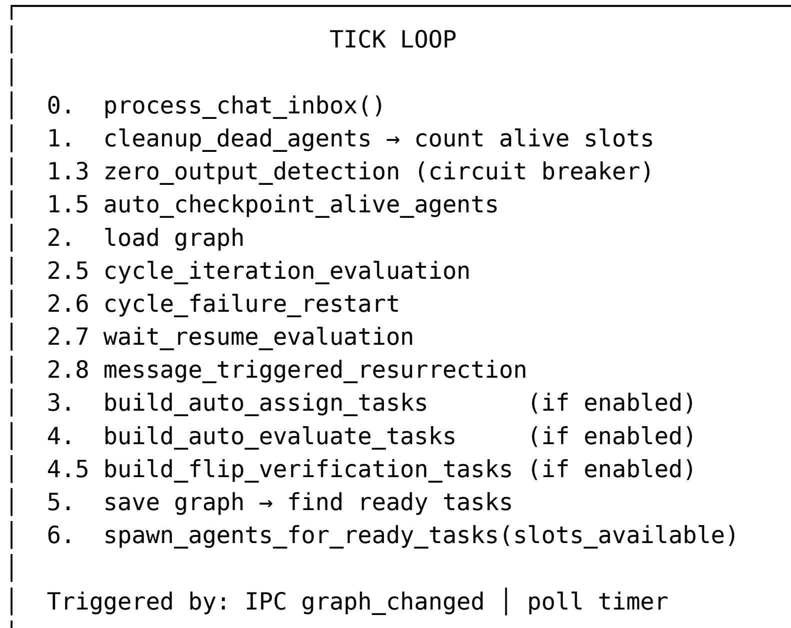
Phase 3: Build auto-assign meta-tasks. If `auto_assign` is enabled in the agency configuration, the coordinator scans for ready tasks that have no agent identity bound to them. For each, it creates an `assign-{task-id}` meta-task that the original task is after. This meta-task, when dispatched, will 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.

Phase 4: Build auto-evaluate meta-tasks. If `auto_evaluate` is enabled, the coordinator creates `evaluate-{task-id}` meta-tasks that are after 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 run`, 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.

Phase 4.5: FLIP verification. If `flip_verification_threshold` is configured, the coordinator scans for tasks with FLIP scores below the threshold and creates `.verify-flip-{task-id}` verification tasks dispatched to a stronger model (Opus by default). FLIP (Fidelity via Latent Intent Probing) is an independent fidelity check that reconstructs what the task prompt must have been from the agent's output alone, then scores the match—see Section 6 for details.

Phase 5: Save graph and find ready tasks. If previous phases modified the graph (adding meta-tasks, adjusting dependencies), 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. A global zero-output backoff check also runs here: if the backoff is active (from phase 1.3), spawning is skipped entirely.

Phase 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

  0.  process_chat_inbox()
  1.  cleanup_dead_agents → count alive slots
  1.3 zero_output_detection (circuit breaker)
  1.5 auto_checkpoint_alive_agents
  2.  load graph
  2.5 cycle_iteration_evaluation
  2.6 cycle_failure_restart
  2.7 wait_resume_evaluation
  2.8 message_triggered_resurrection
  3.  build_auto_assign_tasks       (if enabled)
  4.  build_auto_evaluate_tasks     (if enabled)
  4.5 build_flip_verification_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 10: The 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 and exec-mode. 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`).

The task's `exec_mode` field further controls execution weight: `full` (default—complete tool access), `light` (read-only tools, suitable for analysis and review tasks), `bare` (only `wg` CLI commands, no file editing), or `shell` (no LLM—runs the task's `exec` field directly, like the shell executor). Exec-mode and executor are complementary: the executor determines *which backend* runs the task; exec-mode determines *how much autonomy* the agent has within that backend.

Resolve the model and provider. 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. Provider selection follows the same chain via the `provider` field (`anthropic`,

openai, openrouter, or local). This lets you pin specific tasks to specific models and providers—a cheap model on OpenRouter for routine evaluation tasks, a capable Anthropic model 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.

Resolve the context scope. The coordinator determines how much surrounding context the agent receives by resolving a *context scope* through a priority chain: the task's own context_scope field takes precedence, then the assigned role's default context scope, then the coordinator's configured scope, then the default of task. The four levels are cumulative—each tier includes everything from the tier below:

- **Clean.** Bare executor: the agent receives its identity, the task description, upstream dependency context, and any cycle/loop info. No workflow instructions, no graph patterns, no system awareness. Used for tightly-scoped tasks where extra context is noise.
- **Task.** The standard default. Adds workflow commands (wg done, wg fail, wg log, wg artifact), graph patterns (pipeline, diamond, scatter-gather), reusable function hints, downstream consumer awareness, and the ethos section that encourages autopoietic behavior.
- **Graph.** Adds the project description from config.toml and a 1-hop neighborhood summary showing immediate graph context (neighboring tasks and their statuses).
- **Full.** Adds a system awareness preamble (explaining the agency, cycles, functions), the complete graph summary, and the project's CLAUDE.md content.

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. Skills are resolved at this point: file skills read from disk, URL skills fetch via HTTP, inline skills expand in place. The prompt sections are assembled according to the resolved context scope. The rendered prompt is written to a file in the agent's output directory.

For tasks that are part of a structural cycle, the rendered prompt carries additional context: the current loop_iteration (which pass this is) and a note about the --converged flag. This informs the agent that it can signal wg done <task-id> --converged to stop the cycle early—preventing further iteration even if max_iterations hasn't been reached and guard conditions are met. The "converged" tag is placed on the cycle header regardless of which member the agent completes. The cycle evaluator checks for this tag before re-opening members for the next iteration. This mechanism exists because cycles that run to max_iterations when the work has already stabilized waste compute and agent time. Convergence is the agent's way of saying "the work is stable, no more iterations needed." A subsequent wg retry clears the convergence tag, allowing the cycle to resume.

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

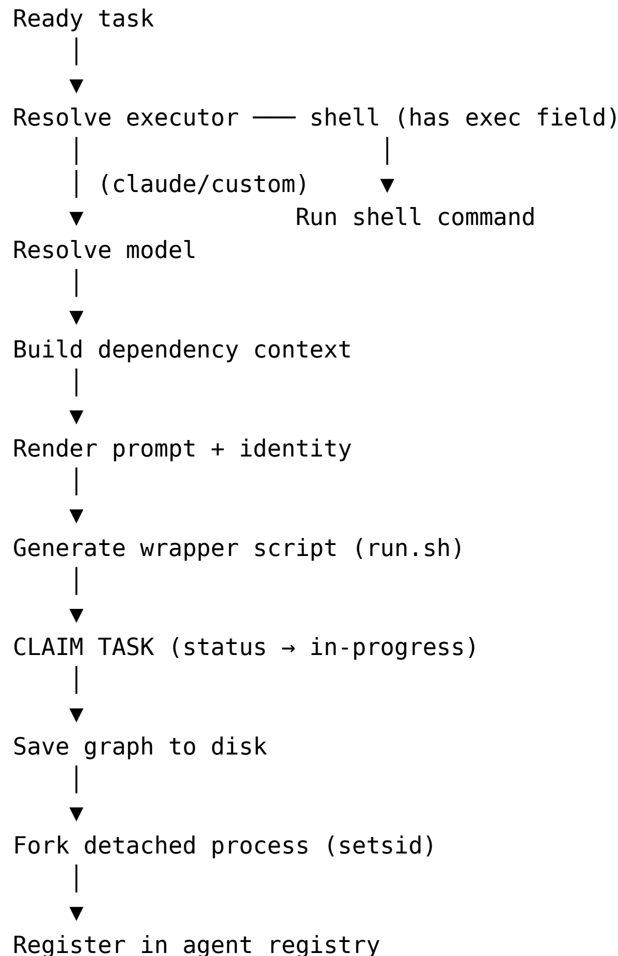
- Unsets CLAUDECODE and CLAUDE_CODE_ENTRYPOINT environment variables so the spawned agent starts a clean session.
- Pipes the prompt file into the executor command (e.g., cat prompt.txt | claude --print --verbose --output-format stream-json).

- 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 11: 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 is before 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 are before a single aggregator: the aggregator only becomes ready when all its predecessors are terminal, and then a single agent handles the synthesis.

These patterns are not built-in primitives. They emerge from dependency edges. 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 that is after 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 run <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 evolution (see §5).

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 predecessor 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.

Evaluations created by auto-evaluate carry a source field set to "llm", identifying them as internal assessments from the LLM evaluator. External evaluations can be recorded via wg evaluate record --task <id> --source <tag> --score <0.0-1.0>, where the source tag is a freeform string—"outcome:sharpe", "ci:test-suite", "vx:peer-123", or any label meaningful to the project. The evolver reads all evaluations regardless of source (see §5), enabling it to weigh internal quality assessments against external outcome data when proposing improvements to the agency.

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 cycle iteration is 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 Multi-Coordinator Sessions

A single service daemon can host multiple coordinator sessions, each managing an independent scheduling context. This enables parallel workstreams within the same project—for example, one coordinator handling feature development while another handles maintenance tasks.

Coordinator sessions are managed via service subcommands:

- `wg service create-coordinator` creates a new session.
- `wg service stop-coordinator` stops a running session (kills its agent and resets to open).
- `wg service archive-coordinator` archives a completed session (marks it done).
- `wg service delete-coordinator` removes a session entirely.

The maximum number of concurrent coordinators is configured via `wg config --max-coordinators`. The `wg chat --coordinator <ID>` flag targets messages to a specific coordinator session.

5.11 Compaction, Sweep, and Checkpoint

Three maintenance commands support long-running projects:

Compaction. `wg compact` distills the current graph state into a condensed summary file (`context.md`), providing a snapshot of project status, recent decisions, and key patterns. Within the service daemon, compaction runs as the `.compact-0` task—a structural cycle where the coordinator periodically introspects its own state and produces a compressed context for future agent prompts.

Sweep. `wg sweep` detects orphaned in-progress tasks—tasks claimed by agents whose processes have died without triggering normal cleanup. It scans the agent registry, checks PIDs, and offers to reclaim or reset affected tasks. Sweep is a manual recovery tool for cases where the coordinator's normal dead-agent detection misses something (e.g., after a system reboot).

Checkpoint. `wg checkpoint` lets a running agent save a progress snapshot during a long-running task. If the agent is interrupted—OOM-killed, timed out, or manually stopped—a replacement agent can resume from the checkpoint rather than starting from scratch. Checkpoints are stored alongside the task's artifacts and injected into the recovery context.

5.12 Service Restart

`wg service restart` performs a graceful stop-then-start cycle. Running agents continue undisturbed (they are detached processes), but the coordinator re-reads configuration and starts fresh. This is the standard way to pick up configuration changes that `wg service reload` cannot apply.

5.13 Observing the System

The IPC protocol lets tools talk to the daemon. But many integrations need to observe the graph from the outside—a CI system that triggers on task completion, a dashboard that tracks agent progress, a portfolio manager that records outcomes. For these, workgraph provides `wg watch`.

`wg watch` streams a real-time event feed of graph mutations to standard output. Each line is a JSON object with a type, timestamp, optional task ID, and a data payload carrying the operation detail. The event types mirror the operations log: `task.created`, `task.started`, `task.completed`, `task.failed`, `task.retried`, `evaluation.recorded`, `agent.spawned`, `agent.completed`. The stream reads from the same prove-

nance log that records every mutation to the graph—`wg watch` is not a separate event system but a live tail of the log with structured formatting.

Events can be filtered. The `--event` flag accepts categories—`task_state` for all task transitions, `evaluation` for scoring events, `agent` for spawn and completion. The `--task` flag narrows to events affecting a specific task by ID prefix. These filters compose: you can watch only state-change events for tasks in a particular subtree. The `--replay N` flag emits the last N historical operations before switching to live streaming, letting a newly launched adapter catch up on recent history without scanning the full log.

5.13.1 The Adapter Pattern

`wg watch` is one side of a broader integration architecture. External systems interact with workgraph through five ingestion points, each corresponding to a different kind of information flow:

Point	Command	What flows
Evaluation	`wg evaluate record`	Scores with source tags — external outcome data enters the agency's performance records.
Task	`wg add`	New work items — an external system can inject tasks with dependencies, skills, and descriptions.
Context	`wg trace import`	Peer exports and knowledge artifacts — enriching agent prompts with cross-boundary data.
State	`wg done`, `wg fail`, `wg log`	Status changes and progress events — an external system can mark work complete or record observations.
Observation	`wg watch`	The event stream *out* — external systems observe what is happening without polling.

The generic adapter follows a four-step pattern: *observe* the graph via `wg watch`, *translate* external data into workgraph's vocabulary, *ingest* via the appropriate CLI command, and *react* by triggering external actions. A CI adapter might observe `task.completed` events, run a test suite, and record the result via `wg evaluate record --source "ci:tests"`. A portfolio manager might observe agent completions, measure real-world outcomes, and feed scores back as external evaluations. The adapter pattern is deliberately simple—each integration is a small loop of observe, translate, ingest, react—because the ingestion points are stable CLI commands, not a bespoke API.

5.13.2 The Operations Log and Trace

Every mutation to the graph—task creation, status change, evaluation, agent spawn—is recorded in the operations log (`operations.jsonl`). This log is the raw material for both `wg watch` (live streaming) and `wg trace` (historical reconstruction). The coordinator does not maintain a separate event bus; `wg watch` simply tails the operations log and formats each entry as a typed JSON event.

The trace system builds on this foundation. `wg trace show` reconstructs the history of a task or subtree by reading the operations log and replaying state transitions. `wg trace show --animate` takes this further: it reconstructs temporal snapshots of the graph at each mutation, then plays them back in the terminal as an interactive animation—tasks transitioning between statuses

over time, a visual record of how work flowed through the graph. You can pause, step forward and backward through snapshots, and adjust playback speed.

`wg trace export --visibility <zone>` produces a filtered, shareable snapshot of the trace. The visibility parameter controls what crosses organizational boundaries: `internal` exports everything, `public` sanitizes the export (task structure without agent output, logs, or evaluations), and `peer` provides richer detail for trusted peers (including evaluations with notes stripped). The corresponding `wg trace import` ingests a peer's export, namespacing imported tasks to avoid ID collisions and tagging evaluations with their origin for provenance tracking. These exports use the `visibility` field on each task (see §2) to determine what is included at each zone level.

These capabilities—watch, trace, export, import—form a layered system. The operations log is the ground truth. The watch stream is its real-time face. The trace commands are its analytical tools. And the export/import mechanism is how organizational memory crosses boundaries.

5.14 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.

5.15 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.

5.16 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 predecessors 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 (called *tradeoffs* in the CLI—`wg tradeoff`). 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, 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.

6.1.3 FLIP: Fidelity via Latent Intent Probing

Standard evaluation asks an LLM to read the task and its output and score quality. FLIP asks a different question: *does the output faithfully reflect what was asked?*

The FLIP pipeline has two phases. In the *inference* phase, an LLM (default: Sonnet) receives only the agent's output—no task description—and reconstructs what the original prompt must have been. In the *comparison* phase, a second LLM (default: Sonnet) scores how well the reconstructed prompt matches the actual task description. The result is a fidelity score: high FLIP means the output clearly addresses the task; low FLIP means the output may have drifted from the intent, even if it looks competent in isolation.

FLIP runs alongside standard evaluation when `flip_enabled` is true in the agency configuration. The scores are recorded with source `"flip"` and propagate through the same three-level mechanism as standard evaluations.

When `flip_verification_threshold` is configured, tasks with FLIP scores below the threshold automatically receive a `.verify-flip-{task-id}` verification task. This verification task is dispatched to a stronger model (Opus by default) to independently confirm or reject the result. The full agency pipeline is: **evaluate → FLIP → verify → evolve**.

FLIP is configured via `wg config`:

```
wg config --flip-enabled true
wg config --flip-inference-model sonnet
wg config --flip-comparison-model sonnet
wg config --flip-verification-threshold 0.7
wg config --flip-verification-model opus
```

6.1.4 The eval-gate mechanism

The evaluation gate is a quality floor. When `eval_gate_threshold` is configured, any evaluated task whose weighted score falls below the threshold is automatically *rejected*—its status is set to failed with a descriptive reason citing the score and threshold. This prevents low-quality work from being accepted and unblocking downstream tasks.

By default, the eval gate applies only to tasks tagged `"eval-gate"` (tasks created with `--verify` receive this tag automatically). Setting `eval_gate_all` to true extends the gate to *all* evaluated tasks, creating a project-wide quality minimum.

```
wg config --eval-gate-threshold 0.7
wg config --eval-gate-all true
```

The eval gate runs as the final step of `wg evaluate run`, after scoring is complete and before the evaluation is considered finished. A rejected task can be retried (`wg retry`), which re-opens it for a fresh attempt with recovery context from the failed evaluation.

6.1.5 External evaluation sources

Not every signal about an agent's performance comes from an LLM reading its output. A trading agent might produce clean, well-structured code that scores 0.91 on internal evaluation —and lose money. A documentation agent might produce prose that the evaluator loves but that users find confusing. Internal quality assessment is necessary but not sufficient. The real test is what happens when the work meets the world.

Every evaluation carries a `source` field that identifies where the score came from. The internal auto-evaluator writes `source: "llm"`. External evaluations use freeform tags that name their origin: `"outcome:sharpe"` for a portfolio's realized Sharpe ratio, `"ci:test-suite"` for a continuous integration result, `"vx:peer-123"` for a score received from a federated peer, `"user:feedback"` for a human's direct assessment. The tag is a string, not an enum—any external system can define its own source convention.

External evaluations enter the system through `wg evaluate record`:

```
wg evaluate record --task portfolio-q1 \
  --source "outcome:sharpe" --score 0.72 \
  --notes "Realized Sharpe below target"
```

The command requires a task in done or failed status, resolves the agent identity from the task's assignment, and writes the evaluation to the same store as internal evaluations. It propagates to the same three levels—agent, role with context, motivation with context. From the perspective of the performance records, an external evaluation is indistinguishable from an internal one except for the source tag.

This is where the evolutionary signal becomes rich. Consider an agent that scores 0.91 internally (clean code, complete deliverables, good style) but 0.72 on outcome (the code it wrote performed poorly in production). The evolver sees both scores in the performance summary. The gap between internal quality and external outcome is itself a signal—it suggests the role's desired outcome or the motivation's trade-offs need to account for domain-specific success criteria, not just code quality. The evolver can propose a mutation that sharpens the role toward outcomes the internal evaluator cannot see.

The five dimensions of external signal that can flow into a workgraph project—evaluation scores, new tasks, imported context, state changes, and event observations—form the system's interface with its environment. Evaluation is the most direct: it converts external reality into the same currency the evolver already reads.

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 run`. This is a deliberate design choice. The system accumulates evaluation data automatically (via the coordinator's auto-evaluate feature), 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. Deferred operations (such as self-mutations requiring human review) are managed via `wg evolve review`.

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. Two additional configuration options—`creator_agent` and `creator_model` (set via `wg config --creator-agent` and `wg config --creator-model`)—control the provenance metadata recorded on entities the evolver creates. When set, newly created roles, motivations, and agents record these values, providing a traceable link between evolutionary output and the identity and model that produced it.

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.

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.

Tradeoff tuning. Adjust the trade-offs on an existing tradeoff (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 tradeoff'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 run --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 run` executes, the following sequence runs:

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 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 run --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 tradeoff 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 tradeoff 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.

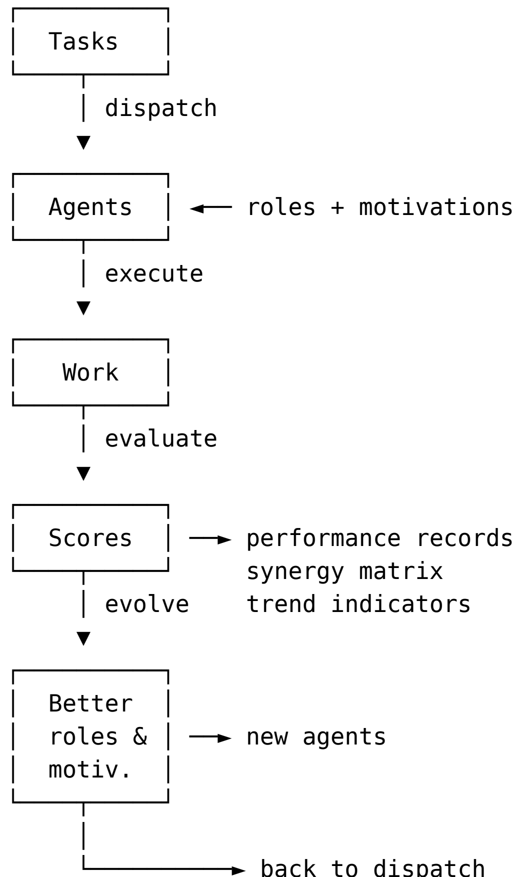
6.6 The Autopoietic Loop

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

Work enters the system as tasks. The coordinator dispatches agents—each carrying an identity composed of a role and a motivation—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.

6.6.1 Federation: cross-organizational learning

The autopoietic loop described above is closed within a single project. Federation opens it.

Agency entities—roles, motivations, agents, and their evaluation histories—can be shared across workgraph projects via `wg agency pull` and `wg agency push`. Named remotes point to other projects' agency stores. When evaluations are transferred, they merge with local performance records: duplicates are identified by task ID and timestamp, and average scores are recalculated from the combined set. Content-hash IDs make this natural—an entity with the same identity-defining content has the same ID in every project, so deduplication is automatic.

What this means for evolution is concrete. A role that has been evaluated across three projects carries a richer performance record than one evaluated in a single project. The evolver sees a broader sample. A role that scores well on code tasks in one project but poorly on documentation tasks in another presents a clearer picture than either project could provide alone. Federation does not change the evolutionary mechanisms—it enriches the data they act on.

The sharing boundary is controlled by task visibility. Every task carries a `visibility` field: `internal` (the default—nothing crosses organizational boundaries), `public` (sanitized for open sharing—task structure without agent output or logs), or `peer` (richer detail for trusted peers—includes evaluations and workflow patterns). Trace exports (`wg trace export --visibility <zone>`) filter according to this field. The result is a structured, shareable view of work product—enough for a peer to learn from without exposing internal operational detail.

6.6.2 Functions: organizational routines

When a workflow pattern proves effective—a plan-implement-validate cycle that consistently produces high evaluation scores—it can be extracted into a reusable template. `wg func extract` reads the completed task graph, captures the task structure, dependencies, structural cycles, and agent role hints, and writes a parameterized function to `.workgraph/functions/`. `wg func apply` creates a fresh task graph from that template with new inputs.

These functions are the system's organizational routines—the term Nelson and Winter (1982) used for the regular, predictable patterns of behavior that serve as an organization's institutional memory. A routine extracted from a successful feature implementation captures not just what tasks to create, but what skills to require, what review loops to include, and what convergence patterns to expect. It is heritable (shareable across projects via the same YAML format), selectable (routines that produce good evaluation scores are retained; others are revised or abandoned), and mutable (a human or an LLM can edit the template to adapt it).

The connection to evolution is direct. Functions capture workflow structure. Evolution improves the agents that execute those workflows. Together, they represent two axes of organizational

improvement: better processes and better performers. A well-extracted function dispatched to well-evolved agents is the system's equivalent of a mature team following a proven playbook.

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, tradeoffs with constraints that are too strict—use a targeted strategy. `--strategy mutation` for improving existing entities. `--strategy tradeoff-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 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.

Mix internal and external signals. Do not evolve on internal evaluations alone if external outcome data is available. Record CI results, production metrics, or user feedback via `wg evaluate record --source <tag>`. The evolver is most effective when it sees both "the code was well-written" and "the code worked in practice"—the gap between the two is where the most useful mutations live.

Pull before evolving. If you maintain multiple workgraph projects or collaborate with peers, run `wg agency pull` before `wg evolve`. Federation imports evaluation data from remote stores, giving the evolver a broader performance picture. A role evaluated across three projects is a more reliable signal than one evaluated in one.

Extract routines from success. When a workflow pattern produces consistently high scores, extract it with `wg func extract`. The resulting function preserves the structure that worked. Combine this with evolution: evolve the agents, keep the proven process.