1 Agent Skill Architecture

Complex scientific workflows demand agents that not only execute multi-step tasks but also learn from their own execution. A static collection of capabilities cannot anticipate every failure mode, adapt to new domains, or close quality gaps discovered only at runtime. We address this challenge with a meta-skills framework built on two pillars: a *three-layer invocation hierarchy* that separates routing, orchestration, and atomic execution; and a *closed-loop evolution mechanism* that transforms every task execution into a learning opportunity.

Three-layer invocation model. The system organizes capabilities into three layers of increasing granularity (Figure 1). *Entry commands* (L0) are thin routers that accept natural-language requests—three commands (`/meta`, `/build`, `/research`) constitute the entire user-facing interface. *Path templates* (L1) are declarative YAML recipes specifying step sequences, gate requirements, and branch conditions; eight templates cover workflows from skill health audits to hypothesis-driven research. *Atomic capabilities* (L2) are 29 contract-driven building blocks with typed inputs and outputs, organized across an eight-stage lifecycle (Discover through Knowledge). Cross-cutting *quality policies* inject verification rules automatically at gate transitions based on artifact types, decoupling what the agent can do, how it sequences actions, and when it enforces quality. A visibility boundary hides all internal layers behind `_`-prefixed directories; users see only three commands.

Design principles. Two ideas enable the system to grow without destabilizing. First, components are separated by *change rate*: L0 commands are frozen, L2 capabilities evolve rapidly in hidden directories, and policies evolve slowly as failure modes are codified—additions to one layer require no coordinated changes to others. Second, constraints are enforced *mechanically*: naming governance (18 controlled verbs, 91 canonical objects) is checked by validation scripts, and quality gates fire at every transition regardless of whether the agent "remembers" to invoke them.

Continuous self-evolution. The architecture's central contribution is a closed loop that makes the system self-improving (Figure 2). Every L0 command defaults to a dual-agent team: an *executor* performs the user's task while an *observer* (running on a cheaper model) monitors for improvement signals—policy violations, capability gaps, step deviations, repeated errors, quality misses, and automation candidates. Overhead remains low ($\sim$1.05$\times$ for simple tasks, 1.3–1.5$\times$ for complex ones) through a complexity tiering system that adjusts observer depth. Each finding maps to a concrete change: a new L2 capability, an L1 path adjustment, or a new `rule-*` policy.

Quality policies function as a *one-way ratchet*: once a failure mode is identified, it is formalized into a policy enforced mechanically at every subsequent execution, preventing the same class of error from recurring. Cross-session continuity relies on a three-layer memory protocol—an auto-injected summary (L1), a structured handoff document (L2), and a permanent session archive (L3)—all updated before session termination. Observer reports feed into a recurring-patterns index that elevates repeated signals for systematic resolution, closing the loop from execution back to architecture. The eight-stage lifecycle further supports a hypothesis-driven research pipeline (`/research`) with three feedback loops—fail-to-fix, iterate, and pivot—and a persistent hypothesis tree that accumulates results across experiments.

A Agent Architecture Details

This appendix provides detailed descriptions and figures for the agent skill architecture presented in Section 1.

A.1 Three-Layer Invocation Model

The system organizes agent capabilities into three layers of increasing granularity (Figure 1). At the top, *entry commands* (Layer 0) serve as thin routers: three commands—`/meta`, `/build`, and `/research`—accept natural-language requests and match them to appropriate workflows. Below these, *path templates* (Layer 1) are declarative YAML recipes that specify step sequences, gate requirements, and branch conditions for iterative loops. The core system defines eight such templates covering workflows from skill health audits to hypothesis-driven research. At the base, *atomic capabilities* (Layer 2) comprise 29 contract-driven building blocks, each with typed inputs and outputs, organized across an eight-stage lifecycle: Discover, Decide, Build, Verify, Deliver, Operate, Review, and Knowledge.

Cross-cutting quality policies operate independently of this hierarchy. Rather than relying on agent judgment to decide when quality checks should run, the system injects verification rules automatically at gate transitions based on the *artifact types* each step produces. A step that emits an evidence bundle triggers reproducibility checks; a step that produces a gate verdict invokes minimum-quality thresholds. This artifact-driven design decouples *what* the agent can do (capabilities), *how* it sequences actions (path templates), and *when* it enforces quality (policies), enabling each dimension to evolve independently.

A visibility boundary separates the user-facing interface from internal complexity: users interact with three entry commands, while the 29 capabilities, 8 path templates, and 9 quality policies reside in directories prefixed with _, hidden from auto-discovery.

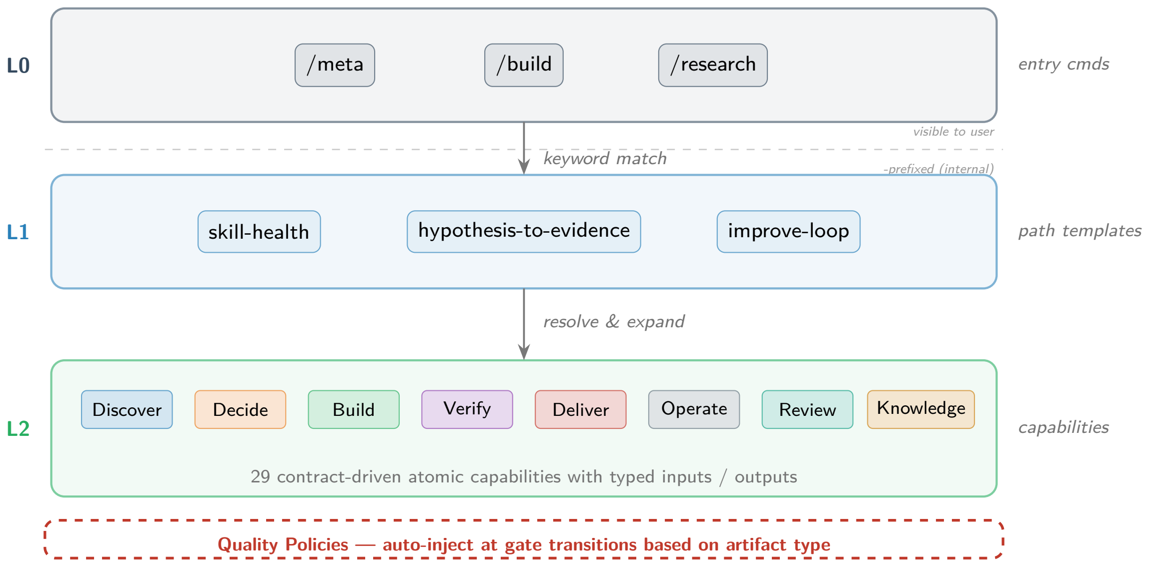

Figure 1: Three-layer skill invocation architecture. Entry commands (L0) route user requests to path templates (L1), which expand into atomic capabilities (L2) organized by lifecycle stage. A dashed visibility boundary separates the three user-facing commands from internal layers prefixed with _. Quality policies inject automatically at gate transitions based on the artifact types produced.

A.2 Design Principles

Two foundational ideas shape the architecture and enable the system to grow without destabilizing.

Separate components that change at different rates. Entry commands (L0) are frozen and rarely modified. Atomic capabilities (L2) evolve rapidly as the system encounters new task types, and reside in hidden directories where changes cannot disrupt the user interface. Quality policies sit between these extremes, evolving slowly as new failure modes are codified. This *stability gradient*—frozen foundation, growing middle layer, slowly-evolving guardrails—mirrors the inner/outer loop pattern from DevOps and the fixed-foundation / dynamic-capabilities / safety-guardrails structure observed in self-evolving agent architectures. Because each layer changes at its own rate, additions to one layer do not require coordinated changes to the others.

Enforce mechanically, not by documentation. The system treats the execution environment as a *harness* rather than a set of guidelines that agents may or may not follow. Naming governance (18 controlled verbs, 91 canonical objects) is enforced by validation scripts, not by style guides. Structural checks catch malformed capabilities before deployment. Cross-cutting policies inject quality gates at every gate transition automatically. This design makes violations structurally impossible rather than merely discouraged: a naming error triggers a linter failure, and a missing quality check fires regardless of whether the agent "remembers" to run it.

A.3 Continuous Self-Evolution

The architecture's central contribution is the closed loop that makes the system *self-improving*: every skill execution feeds observations back into the system's own capabilities, policies, and knowledge (Figure 2). Four mechanisms form this loop.

Accompanied execution. Every L0 command defaults to a dual-agent team: an *executor* that performs the user's task, and an *observer* that monitors for improvement opportunities. The observer runs on a smaller, cheaper model (Haiku) and employs signal-based monitoring rather than full-trace analysis. This keeps overhead low: approximately 1.05× for simple tasks (read, search, check) and 1.3–1.5× for complex ones (write, build, research). A complexity tiering system classifies commands into *light* and *full* tiers, with observers adapting their depth accordingly—light-tier observers perform a single post-hoc analysis, while full-tier observers monitor three checkpoints aligned with lifecycle gate transitions.

The observer watches for six high-value signal types: policy violations, capability gaps, step deviations, repeated errors, quality misses, and automation candidates. Each finding maps to a concrete change: a new L2 capability, an L1 path adjustment, or a new quality policy. Graceful degradation ensures that observer failures never block the user's task.

Policy ratchet. Quality policies function as a one-way ratchet. Once accompanied execution identifies a failure mode, it is formalized into a new `rule-*` policy and enforced mechanically at every subsequent execution. Because policies trigger based on artifact types rather than agent memory, they prevent regression deterministically—the same class of error cannot recur once the corresponding policy is in place. Over time, the set of enforced policies grows monotonically, progressively closing quality gaps.

Three-layer memory. Cross-session continuity relies on a write-on-end protocol with three persistence layers. Layer 1 is an auto-injected compact summary (`MEMORY.md`) that every new session receives mechanically, requiring no voluntary action. Layer 2 is a structured handoff document (`HANDOFF.md`) read at session start, containing active TODOs, blockers, and recently modified files. Layer 3 is a permanent session archive with timestamped records available for historical lookup. All three layers must be updated before any session terminates; omitting any one constitutes a memory discontinuity. Automated pruning retains the 30 most recent sessions to prevent unbounded growth.

Knowledge capture and pattern accumulation. The Review stage of the lifecycle extracts reusable principles from retrospectives and archives them as knowledge cards. Observer reports feed into a recurring-patterns index (`PATTERNS.md`) that tracks signal types, frequencies, and resolution status across sessions. When the same signal type recurs—for example, repeated step deviations in a particular path template—the index elevates it for systematic resolution rather than one-off repair. This trend-detection mechanism transforms individual observations into structural improvements, closing the loop from execution back to architecture.

A.4 Research Pipeline Integration

The eight-stage lifecycle supports a hypothesis-driven research pipeline instantiated through the `/research` command. The pipeline transforms a research question into validated evidence through three explicit feedback loops. Quality-gate failure returns to Build for implementation fixes. A "continue" decision loops to Decide for the next iteration. A "change direction" signal retreats to Discover for strategic reformulation. Four properties underpin its effectiveness: the pipeline is *hypothesis-driven* (every experiment targets a falsifiable claim), *MVP-recursive* (each iteration tests one hypothesis with pre-defined acceptance thresholds), *fail-fast* (consecutive failures escalate from fixes to strategic pivots), and *knowledge-accumulating* (results from every experiment—including negative ones—feed into a persistent hypothesis tree).

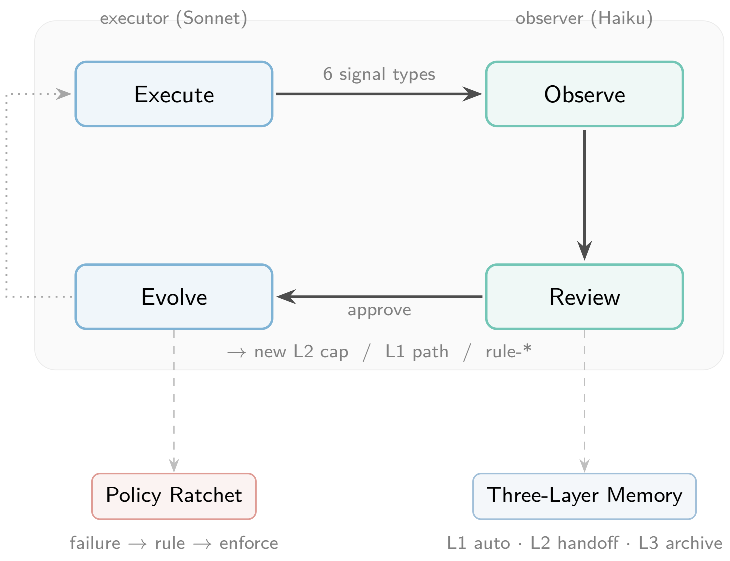

Figure 2: Continuous self-evolution loop. Every skill execution spawns a dual-agent team: an executor performs the task while an observer detects improvement signals. Findings are reviewed and, once approved, fed back as new capabilities, paths, or policies. A policy ratchet mechanically enforces every codified failure mode; a three-layer memory stack ensures cross-session continuity.