PAPER 2 OF 2 · ON THE ANATOMY OF THE SYSTEM THAT RUNS IT.

One Agent, Many Hats

Rules, Skills, and Tools as the Architecture of an Autonomous, Self-Extending Agent System

Sandeep Kavety · July 2026 · Version 1 · 37 min read

ABSTRACT

This paper presents the architecture of the agent system built at Insighter: a single reasoning engine specialised at runtime by three declarative layers. Skills are versioned playbooks that define how the agent works toward a given outcome and allow it to adopt distinct behavioural roles (planner, evaluator, coder, guardian, critic, reflector, communicator) without spawning separate agents. Rules are guardrails enforced at three escalating strengths: as prompt instruction, as coded gates that block progress, and as hard infrastructure boundaries that the model cannot override. Tools are the only actions the agent can take: read-only, tenant-scoped, allowlisted per skill.

Two mechanisms extend this core into an autonomous system. A sandbox lets the agent generate and execute its own code against bounded data snapshots when no named tool fits, under strict isolation, with its output validated before use. A self-extension loop turns experience into capability: lessons distilled from failures and feedback enter future runs through canary; repeated sandbox work and recurring outcomes become proposals for tools and skills, human-promoted into a versioned runtime registry. A layered memory (conversation transcript, rolling user persona, retrieved exchanges, organisational context, and operational learnings) gives the system continuity across runs without retraining.

For known-outcome work, a coded stage machine orders the run and blocks advancement until exit predicates hold. These are hard gates, not a second agent. For presentation artifacts, plan-compose separates judgment (layout and narrative) from deterministic paint (charts and tables

into a host contract), so the model designs while the platform renders. Skills, rules, and tools live in a versioned runtime registry the agent can propose into but never rewrite live.

The thesis of this paper is that an agent system becomes safely autonomous not by removing structure but by making structure declarative, so the system can read it, be bound by it, and, under supervision, grow it.

FOREWORD

> *My earlier working paper on Conversational Decision Intelligence (March 2026) described a methodology for answering natural-language questions over data: plan before acting, reason in steps, recover from failure, learn from feedback. That paper was about the flow of a single conversation. This one is about the anatomy of the system that runs it.*
>
> *In the months since, my work at Insighter has extended the agent layer well beyond question answering: recurring analytical artifacts, composed dashboards, sandboxed computation, and a system that extends its own capabilities with use. Building that required me to answer a question the first paper deferred: what, structurally, is an agent? The default industry answer is "many agents": a planner agent, a critic agent, a coder agent, coordinating through messages. I tried that. It did not hold in production. Coordination overhead grew faster than capability. When an output was wrong, reconstructing which agent decided what meant reconstructing an unauditable message trail.*
>
> *What replaced it is the architecture this paper describes: one agent, many hats. A single reasoning engine whose behaviour is composed, per run and per step, from three declarative layers. Skills tell it how to work; Rules constrain what must never happen; Tools define what actions exist. Roles such as planner, critic, guardian, and coder are not separate agents; they are stances the one agent adopts when its Skills call for them. Around this core are the mechanisms that make autonomy durable across runs, not only within a run: a sandbox where the agent writes and executes its own code, a self-extension loop through which*

> *repeated work hardens into new skills, tools, and rules, and a layered memory that builds a persona of the user and the organisation it serves.*
>
> *This is not a product description and not a survey. It is a set of design choices I tested by building, breaking, and rebuilding under load. The argument I carried from the first paper remains: autonomy is not the absence of structure. It is structure sufficient for unsupervised action within declared bounds.*
>
> Sandeep Kavety
> Hyderabad, India
> 10th July 2026

1. Context and Motivation

1.1 The Question the First Paper Deferred

The conversational methodology of the earlier paper treats the agent as given: a model, a loop, some tools. That is sufficient for a demonstration. In production, three pressures force the question of what the agent actually is:

- **Breadth.** The same system must answer an ad-hoc question, generate a recurring report, review its own narrative for an executive audience, write computation code, and diagnose a stale data source. These demand different behaviours, sometimes within one run.
- **Safety.** The system operates on data that matters, autonomously, without a human watching each step. Its boundaries must hold under model error, prompt injection, and its own creativity.
- **Growth.** Every deployment surfaces work the original design did not anticipate. A system that can only do what it shipped with generates an unbounded engineering backlog; a system that modifies itself without controls generates something worse.

1.2 Why Not a Fleet of Agents

The common answer to breadth is multi-agent: give each behaviour its own agent and let them talk. In production, this pattern failed consistently. Inter-agent messages became an unauditable substrate. When the critic and planner disagreed, the resolution lived in conversation history that could not be reconstructed. State fragmented across agents. Cost multiplied, because every hand-off re-serialised context. Capability did not improve: a "critic agent" is the same model with a different prompt, so the fleet adds coordination cost without new competence.

The alternative inverts the pattern. There is one agent: one loop, one context, one audit trail. Breadth comes from composition: at each step, the engine selects which skill governs, which rules apply, and which tools are permitted, and the agent adopts the role the moment requires. The critic is not an agent; it is a role stance.

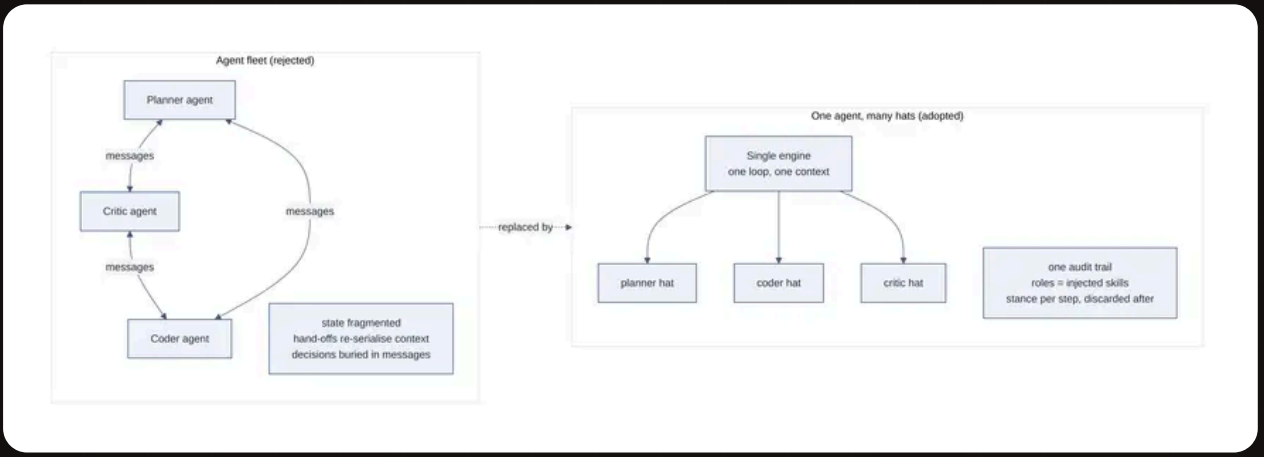


Figure 1.1 — Rejected fleet-of-agents pattern vs. the adopted one-agent, many-hats pattern

1.3 Requirements

The architecture had to satisfy five requirements simultaneously:

- **One engine, many behaviours:** role specialisation without agent sprawl, auditable at every step.
- **Enforceable guardrails:** constraints that hold structurally, not only by instruction.
- **A safe escape hatch:** a way to compute what no named tool covers, without opening the system.
- **Growth under supervision:** new capabilities emerging from use, promoted through review.

- **Continuity:** memory of the user, the organisation, and the system's own mistakes, applied without retraining.

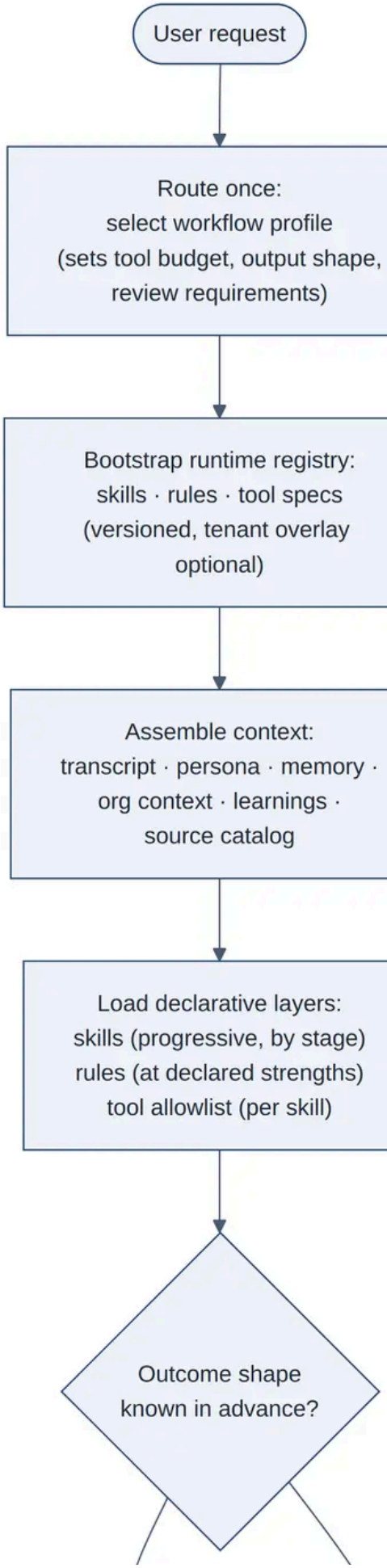
2. The Trinity: Rules, Skills, Tools

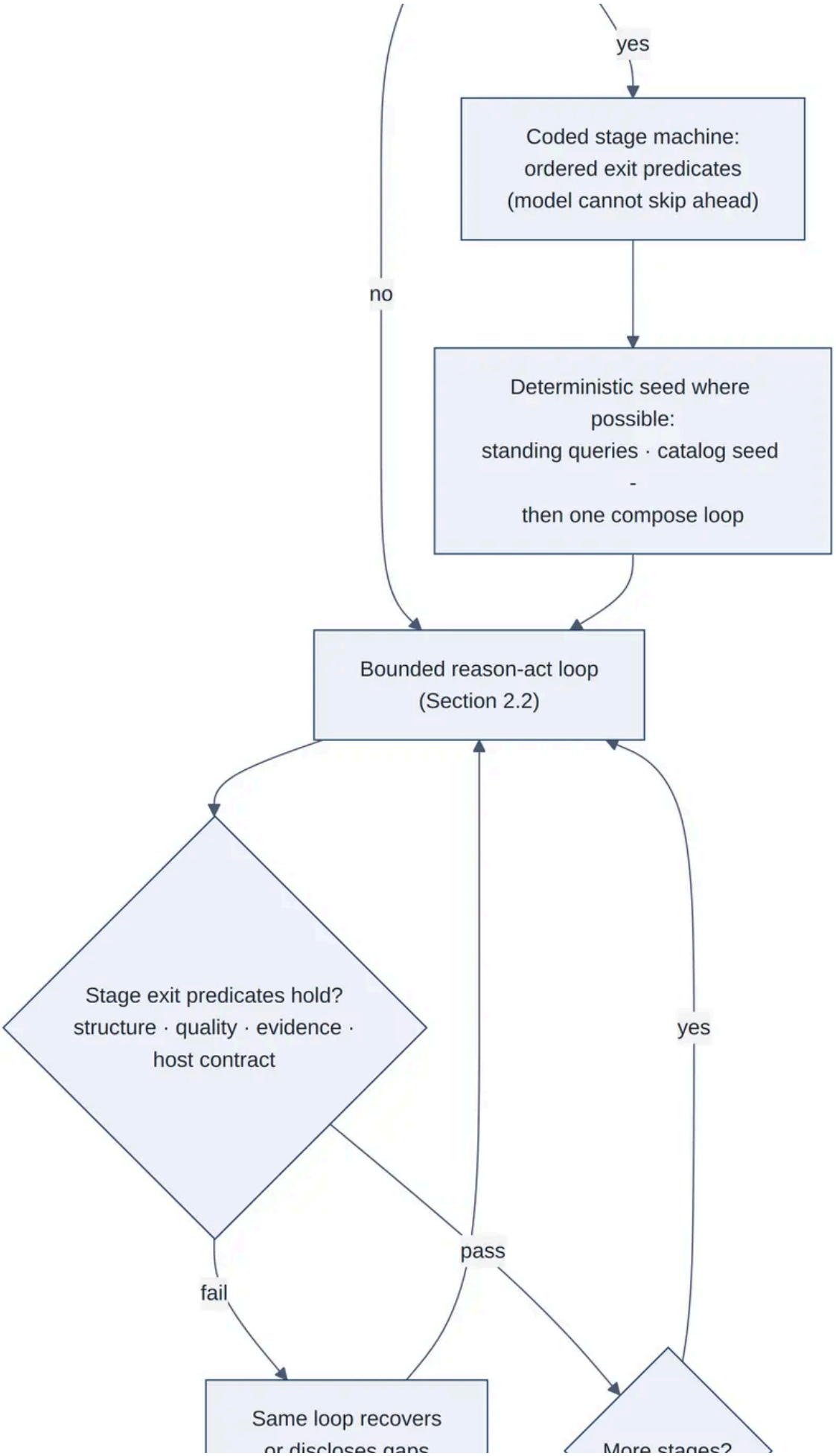
The engine's behaviour is composed entirely from three declarative layers. Each is human-readable, versioned, and reviewable in the same way code is. That property is what everything else in this paper depends on.

Layer	Nature	Question it answers
Skills	Versioned playbooks (structured text)	*How should the agent work toward this outcome?*
Rules	Guardrails with declared enforcement strength	*What must never happen, and how strongly is that held?*
Tools	Typed, executable actions	*What can the agent actually do?*

2.1 Anatomy of a Run

Every request follows the same shape: routed once, composed from the declarative layers, executed as a bounded reason-act loop, gated before delivery, and distilled afterwards.


User request
Route once:
select workflow profile
(sets tool budget, output shape,
review requirements)
Bootstrap runtime registry:
skills · rules · tool specs
(versioned, tenant overlay
optional)
Assemble context:
transcript · persona · memory ·
org context · learnings ·
source catalog
Load declarative layers:
skills (progressive, by stage)
rules (at declared strengths)
tool allowlist (per skill)
Outcome shape
known in advance?


yes
Coded stage machine:
ordered exit predicates
(model cannot skip ahead)
no
Deterministic seed where
possible:
standing queries · catalog seed
-
then one compose loop
Bounded reason-act loop
(Section 2.2)
Stage exit predicates hold?
structure · quality · evidence ·
host contract
yes
pass
fail
Same loop recovers
More stages?

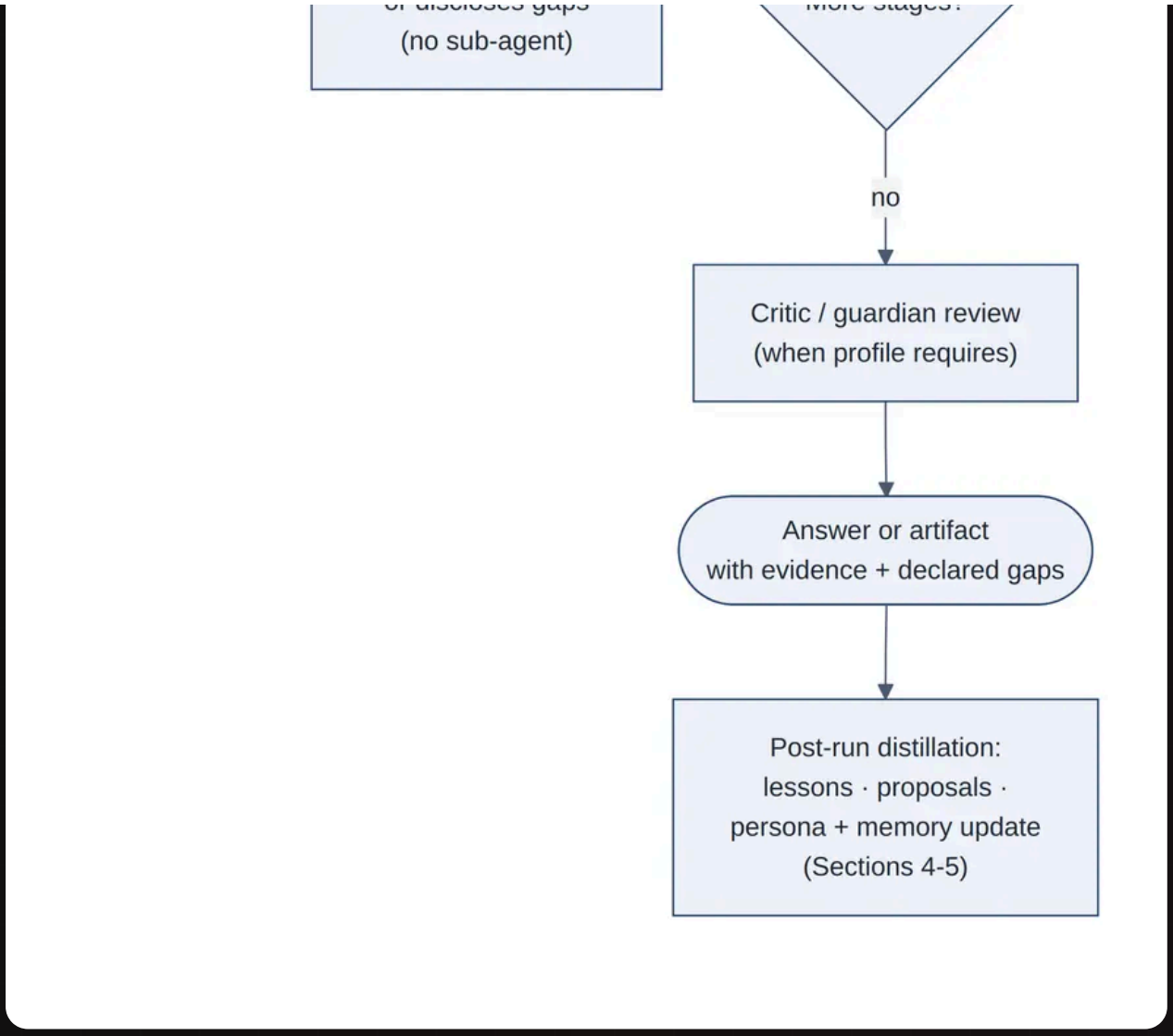


Figure 2.1 — Anatomy of a run, from request to post-run distillation

Two structural choices keep long runs honest.

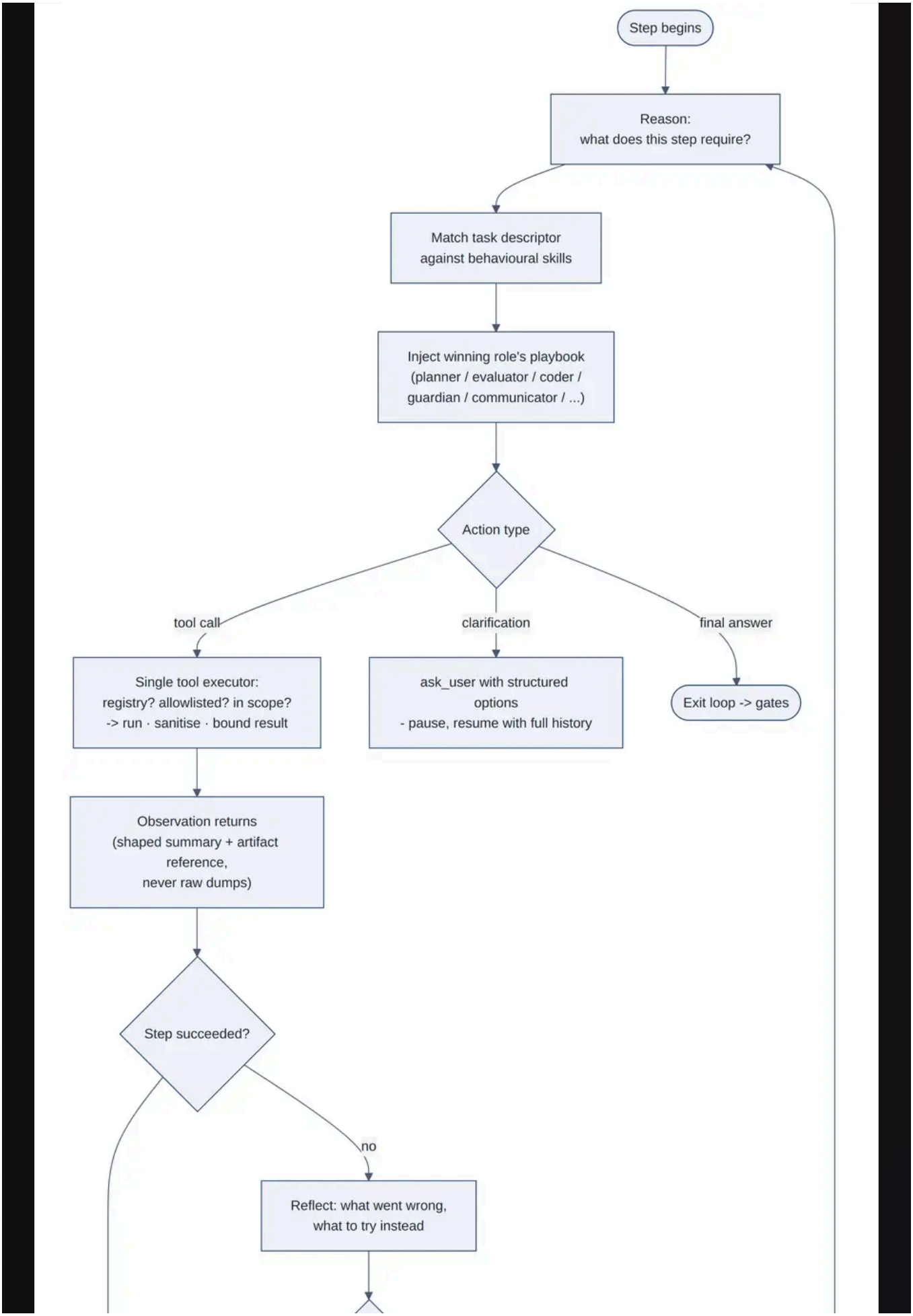
Coded stages over one loop. For known-outcome work (a recurring report, a composed dashboard) the orchestrator advances through an ordered stage machine (intake → discovery → plan → extract → derive → narrate → assemble → quality → recover → present → verify → deliver). Each stage has an exit predicate the model cannot bypass: connector alive, every section planned or gap-documented, findings cited, structure valid, presentation host contract held, numbers reconciled. Stages are gate and telemetry boundaries, not separate agents and not separate transcripts. Early stages are often deterministic or orchestrator-driven; later stages advance when in-loop tools complete and the exit check passes. Recovery is a hat on the same loop (escalating strategy under pinned run scope), not a second agent summoned when quality fails.

Context is still shared. Artifact references (query results live in a store and return by id, not as dumps), structured compaction, and continuation checkpoints keep the window usable, but the agent still reasons over one compacted message array. The design target remains fresh context per phase: each phase ends by writing conclusions, open questions, and artifact ids to a checkpoint the next phase starts from. What shipped is the stage machine and the artifact handoff; what remains open is phase-isolated transcripts. Context occupancy per stage is a metric; a run that nears its ceiling during data work is a routing or skill-design defect made visible, not a silent quality cliff.

Unified seed, then compose. Known-outcome pipelines seed data deterministically (section matrix, catalog fallback) and then enter one orchestrator compose loop. Parallel "question agents" that each own a section were tried and rejected; they reintroduced the fleet's coordination tax without improving attribution.

2.2 The Loop, and How One Agent Wears Many Hats

Inside execution, the agent iterates: reason, adopt a role, act through a tool, observe, validate. The role is selected per step. Before each action, the engine matches the task at hand against the behavioural skills and injects the winning role's playbook into context. The agent about to execute generated code is, at that moment, wearing the guardian's hat; the agent about to write executive commentary is wearing the communicator's. The role is a stance adopted for a step and discarded after. There is no persistent critic with its own memory and agenda, and therefore nothing to reconcile, coordinate, or audit beyond the single loop's own trail.


Step begins
Reason:
what does this step require?
Match task descriptor
against behavioural skills
Inject winning role's playbook
(planner / evaluator / coder /
guardian / communicator / ...)
Action type
tool call
clarification
final answer
Single tool executor:
registry? allowlisted? in scope?
-> run · sanitise · bound result
ask_user with structured
options
- pause, resume with full history
Exit loop -> gates
Observation returns
(shaped summary + artifact
reference,
never raw dumps)
Step succeeded?
no
Reflect: what went wrong,
what to try instead

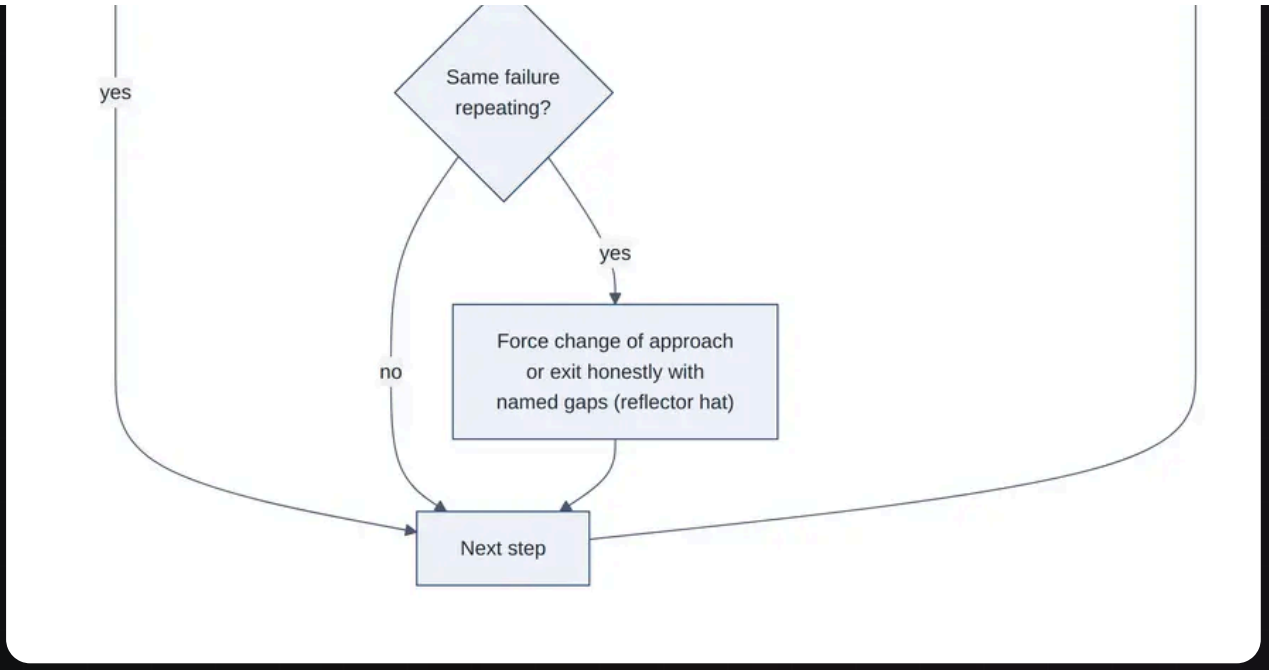


Figure 2.2 — The bounded reason-act loop and how the agent adopts a role per step

One loop action is easy to omit and costly to lack: structured clarification. When the loop cannot proceed without the user (ambiguous intent, a choice between valid interpretations, missing scope) it pauses through a clarification action that presents structured options, and resumes with full history when the user answers. Guessing under ambiguity produces confident errors; asking produces short waits. The pause is a first-class loop state, not an error.

Decision-making remains serial by design; that seriality is the audit trail. Critic, guardian, and reflector passes are hats and tools on the same timeline — they score and return structured verdicts (including backtrack targets), but they do not fork into peer agents. An optional future optimisation is evaluation fan-out: concurrent, stateless judgment calls that only score and never act, with verdicts recorded as observations. Three constraints would keep that from becoming a hidden multi-agent system: fanned-out calls never see each other, never accumulate state across steps, and never act. Until that exists, latency is paid in series; reconstructability is preserved.

Semantic model routing sits beside the hats. Each step is bound to a competence tier (light, standard, or frontier) chosen from the active skill class, tools in play, and message signals, with cost pressure able to downgrade when the window is tight. Synthesis and hard recovery earn the frontier model; extraction and formatting stay cheaper. The model itself is treated as a versioned dependency of the skill library: a regression suite of representative runs gates every model swap, the same way tests

gate a library upgrade. Routing is a coded seam, not a field the model invents mid-run.

The behavioural roles, and when each hat goes on:

Role	Purpose	Typical moment
Orchestrator	Route the run; enforce the allowlist	Run start
Planner	Sequence work by dependency	Multi-step planning
Evaluator	Feasibility, assumptions, confidence	Metric and calculation design
Coder	Minimal, auditable computation scripts	Sandbox generation
Guardian	Adversarial pre-delivery check	After queries, visuals, sandbox output
Critic	Audience-aware narrative review	Before artifact publication
Reflector	Terminal authority on gaps and retries	Loop exhaustion, bounded recovery
Communicator	Plain-language synthesis	Narrative composition
Visual communicator	The right chart, not the impressive one	Visualization steps

2.3 Skills: Progressive Composition

Skills come in four layers that compose per run:

- **Outcome skills** define the target artifact, its structure, its quality bar, its expected sequence of work. Each declares the tools it permits, the step budget it allows, and whether a deterministic seed should run before the loop.

- **Cross-cutting skills** carry the disciplines that apply everywhere: evidence rules, presentation conventions, data-access etiquette.
- **Domain skills** carry what a given industry or function expects to see (indicators, framings, sensitivities), so the engine stays generic while the playbook carries the domain.
- **Behavioural skills** are the roles of Section 2.2.

Skills load by progressive disclosure keyed to the active workflow stage and an outcome funnel, not a fixed three-rung ladder. At run start the engine loads essentials (core discipline, outcome skill, domain when relevant). As the stage machine advances, a funnel selects additional skills whose frontmatter matches the current stage, the active outcome, and optional trigger text (user message and standing instructions). Presentation contracts can load as early as extraction when the outcome will need them later, so context holds what this stage and the near horizon require, not everything the library contains.

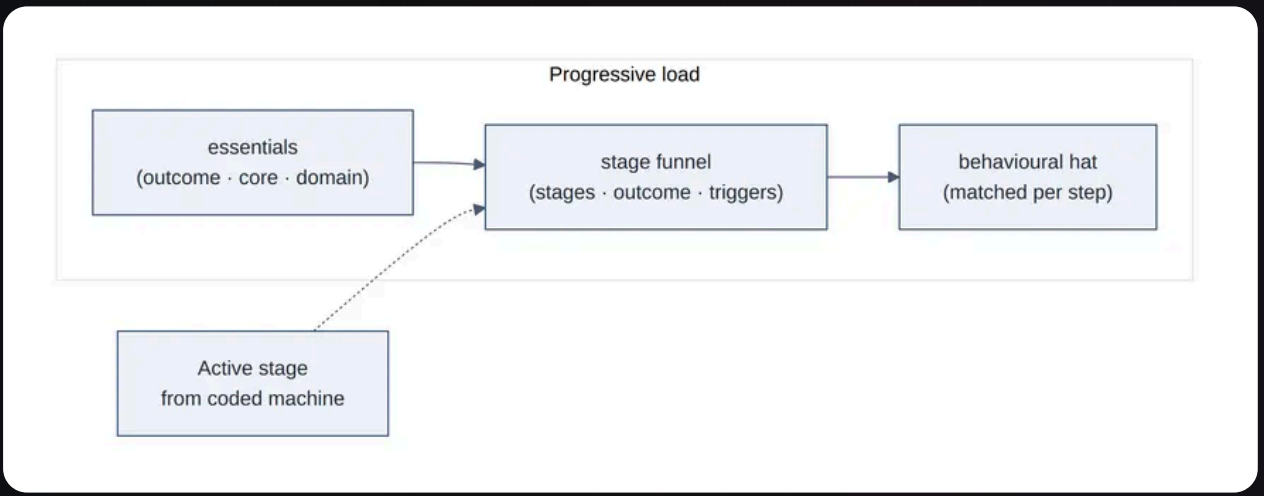


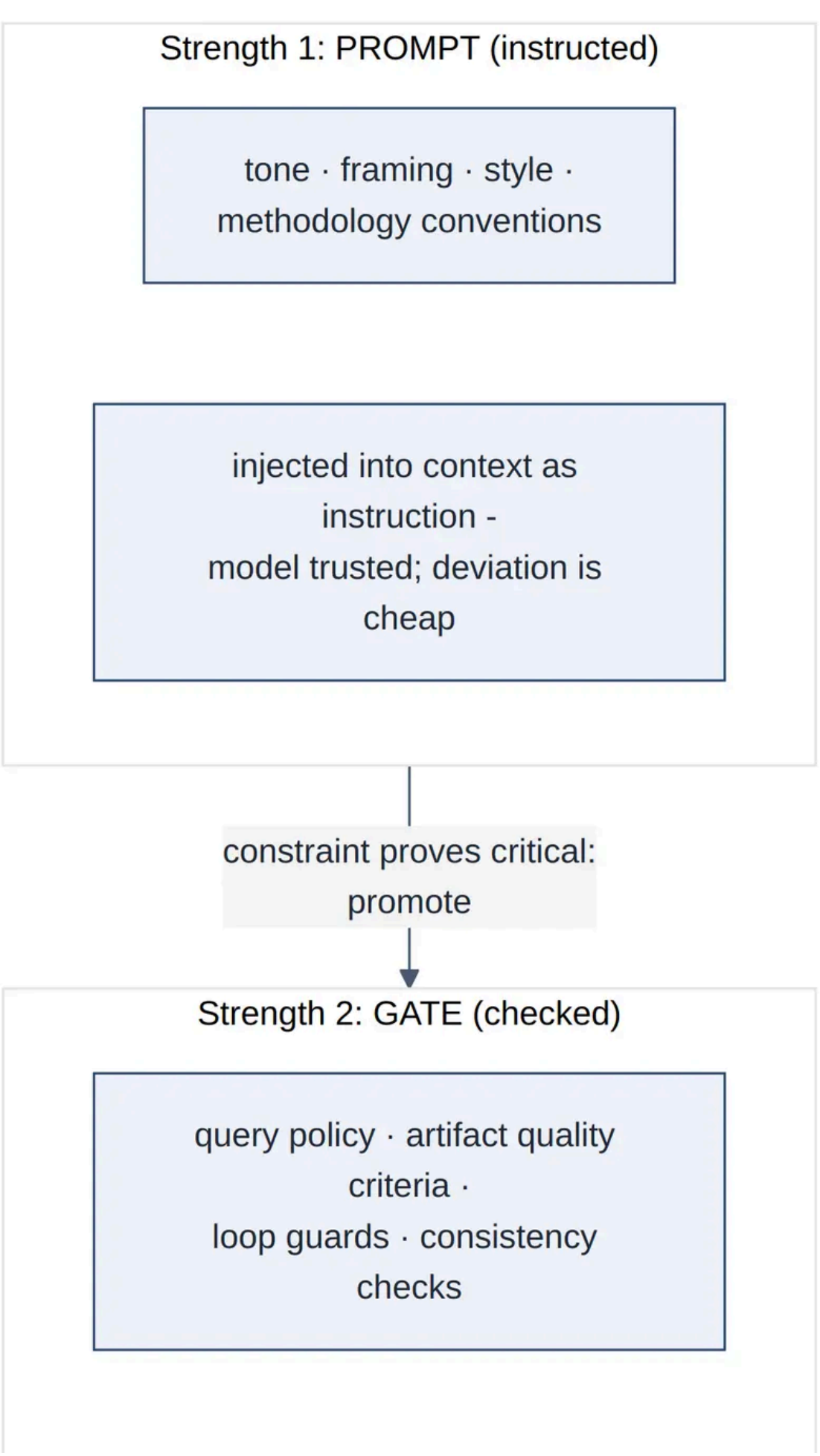
Figure 2.3 — Progressive skill loading keyed to the active workflow stage

Gate-strength rules attach the same way: scoped to stages and outcomes, cited at the trust boundary that enforces them.

2.4 Rules: Guardrails at Three Strengths

A lesson from production incidents: not all constraints deserve the same enforcement. Uniform enforcement fails in both directions: prompt-only enforcement of critical constraints eventually breaks, and hard-coding every

preference makes the system unmaintainable. Rules therefore declare their own enforcement strength, and the engine holds each at the level it declares.


Strength 1: PROMPT (instructed)
tone · framing · style · methodology conventions
injected into context as instruction - model trusted; deviation is cheap
constraint proves critical: promote
Strength 2: GATE (checked)
query policy · artifact quality criteria · loop guards · consistency checks

coded checks at workflow points -
failure blocks progress or forces backtrack;
the model cannot bypass a gate

constraint proves invariant: promote

Strength 3: CODE (structural)

tool allowlist enforcement · tenant scoping · read-only surface · sandbox isolation

held by infrastructure -
true regardless of what the model
generates or is tricked into requesting

Figure 2.4 — Rules held at three escalating enforcement strengths

Rules, like skills, are scoped to a workflow stage, an outcome, or a data source, and every gate cites the rule it enforces, so every blocked action traces to a reviewable document. The promotion ladder also gives growth a path: a constraint can begin life prompt-enforced, prove its importance, and graduate to a gate — governance as a ladder rather than a rewrite.

2.5 Tools: The Only Way to Act

Tools are the agent's entire action surface: querying structured data, searching and reading documents, discovering schemas, visualising, composing dashboards and artifacts, plan-compose presentation (§2.7), deterministic derivation, validation checks, asking the user for clarification, staging skill/rule/tool proposals, and the sandbox trio of Section 3. Every invocation passes through a single executor.

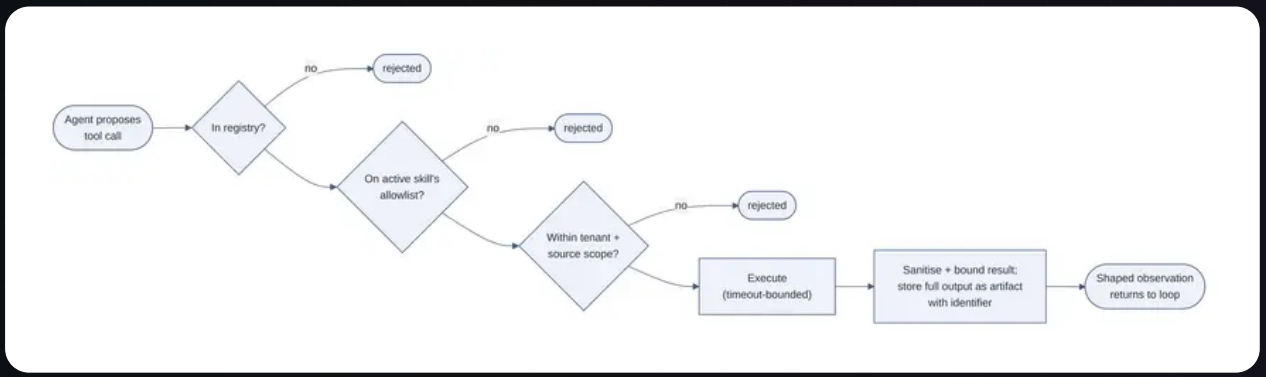


Figure 2.5 — Every tool call passes through a single executor

Two properties are architectural, not incidental. The surface is read-only by construction. That claim needs precision, because the system does write artifacts, transcripts, lessons, and persona snippets (Sections 4–5). Read-only refers to source systems and the outside world: no tool in the registry can write, delete, export, or transmit business data, so the worst case of any agent behaviour is a

wrong answer that shows its work, never a modified record or an exfiltrated one. The system's own writes go only to its tenant-scoped internal stores, through the executor and the post-run pipeline — fixed destinations the model cannot redirect, never through an action the model can aim at an arbitrary target. And validation is itself tooling: consistency checks, output verification, and gap reflection are tools the loop invokes, which means the guardian and reflector roles act through the same audited channel as everything else, returning pass/fail verdicts with explicit backtrack targets rather than informal judgment.

2.6 Reference Designs: What the Three Layers Look Like

The trinity is useful only if each layer can be authored. This section gives one worked example per layer, then shows how the orchestrator composes them at run time. The formats are illustrative; the properties matter more than the syntax. Adapt domain, storage, and naming to your setting.

2.6.1 Designing a Skill

A skill is a versioned document with a machine-readable header and a human-readable playbook. The header is what the orchestrator reads; the playbook is what the model reads. Everything the engine needs to enforce lives in the header; everything the model needs to follow lives in the prose.

```yaml
id: outcome/periodic-performance-report
kind: outcome              # outcome | cross-cutting | domain | behavioural
version: 3                 # bump on every change; runs record which version t
tools:                     # the allowlist: the executor enforces this, not th
  - query_structured_source
  - search_documents
  - derive_metric
  - render_visual
  - compose_artifact
  - propose_presentation_layout
  - compose_presentation
  - check_consistency
  - ask_user
  - propose_skill          # stage drafts only: never live registry writes
```

```
  - propose_rule
  - propose_tool
step_budget: 24             # hard ceiling; reflector takes over at exhaustion
deterministic_seed: true    # known outcome shape -> run predetermined queries
stages:                     # progressive disclosure: funnel attaches this ski
  - query_plan
  - query_execute
  - narrative_quality
  - presentation_publish
review: critic               # none | guardian | critic: what must pass before
---
Periodic Performance Report

Outcome

A recurring report for a named audience: headline movements, drivers,
exceptions, and a forward view. Every figure cites its evidence artifact.

## Shape of the work

1. Seed: run the standing queries for the period (deterministic, parallel).
2. Verify freshness of every source touched; declare staleness, never hide it.
3. Derive period-over-period movements with the derivation tool: never
   arithmetic in prose.
4. For each material movement, find the driver: decompose before you narrate.
5. Compose sections in the audience's order: what changed, why, what to watch.

## Quality bar

- No figure without an artifact reference.
- No driver claim without a decomposition that supports it.
- Unexplained movements are listed as open questions, not smoothed over.

## When to stop

If two decomposition attempts fail to explain a movement, record it as an
open question rather than force a narrative.
```

Design rules that generalise: one outcome per skill (a skill that covers two artifacts will do both badly); the header is contract, the prose is guidance; budgets, allowlists, and review requirements must be enforced by the engine, because prose-only limits are Strength-1 by definition; write the stop condition — the most common skill-authoring failure is describing success and leaving the model to

improvise failure; and version explicitly, because when a run misbehaves, the first question is which version of which skill was loaded.

Competence tiers are not authored as a skill-header field the model can ignore. Semantic routing (§2.2) binds each step to light / standard / frontier from skill class, tools in play, and cost pressure, so the hats remain behavioural while model spend stays a coded seam.

Behavioural skills (the hats) follow the same format but describe a stance rather than an outcome. The guardian's playbook, for instance, is a short adversarial checklist ("assume the previous step is wrong; find how") with a tools list containing only validation tools.

2.6.2 Designing a Rule

A rule declares its own enforcement strength, its scope, and (critically) what enforces it. A rule that names no enforcement point is not enforceable.

```yaml
statement: >
  No unbounded query results ever enter model context. Every structured
  query is limited at source; full results are stored as artifacts and
  only shaped summaries (schema, row count, notable values) return
  to the loop.
strength: gate                        # prompt | gate | code
scope: [data-work]                    # attaches when these stages/tools are ac
enforced_by: executor.result_shaper   # the code path that holds this: named, f
on_violation: block_and_reshape       # gates say what happens, not just what's
history:
  - v1: prompt   # "keep result sets small": violated within a week
```

Three design rules. Match strength to consequence, not to importance. A tone convention is important but its violation is cheap; leave it at prompt strength. Result bounding felt minor but its violation cascades — it earned a gate. The test is *what happens when the model ignores this?*, not *how much do I care?* Name the enforcement point. `enforced_by` turns a rule from documentation into a checkable claim: you can audit that the named code path exists and cites the rule back. Record the promotion history. The `history` field is the governance ladder of Section 2.4 made visible — it tells the next maintainer why this rule sits at the

strength it does, and it is the paper trail for the incidents that put it there. *Note: the current rule documents carry* `statement`, `strength`, *and* `enforced_by`, *but not yet a populated* `history` *— promotion so far has happened in version control rather than in the rule files. The field is the standard for new promotions; backfilling the ledger for existing rules is open work.*

Strength-3 rules look different: they are typically one line of statement and a pointer to infrastructure ("tenant scope is enforced by per-tenant credentials at the data layer"), because their entire content is the enforcement. If you find yourself writing a long strength-3 rule, part of it is probably a gate or a prompt rule wearing the wrong label.

2.6.3 Designing a Tool

A tool is a typed registry entry. The description is prompt engineering (it is how the model decides when to reach for the tool), while the schema and scoping are enforcement.

JAVASCRIPT

```
{
  name: "derive_metric",
  description:
    "Compute a derived value (ratio, growth, share, variance) from values " +
    "in prior query artifacts. Use this for ALL arithmetic: never compute " +
    "in prose. Returns the value with its inputs and formula recorded.",
  input: {
    operation: enum(["ratio", "growth", "share", "variance", "sum"]),
    inputs: array({ artifactId: string, field: string }),  // references, not
    label: string,
  },
  execute: async (input, ctx) => {
    // ctx carries tenant + run scope; the tool cannot be called without it.
    const values = await resolveFromArtifacts(input.inputs, ctx);  // scoped r
    const result = compute(input.operation, values);               // determin
    return {
      summary: `${input.label}: ${result.formatted}`,              // what the
      artifactId: await storeArtifact(result.withProvenance, ctx), // what the
    };
  },
}
```

The properties to copy are these. Inputs are references, not payloads: the model passes artifact identifiers, so data flows through scoped resolution rather than through the model's context, which both bounds context and prevents the model from laundering values it invented. The observation is shaped: the model receives a summary, the full result becomes an artifact with an identifier, and provenance (inputs, formula) is stored with it. Prefer determinism where possible: arithmetic in a tool is testable and citable; arithmetic in prose is neither, which is why the skill above bans it. The description states when to use the tool and when not to, because tool selection is the model's decision and the description is the primary lever over that decision at run time.

Keep the registry small. A small set of well-described, composable tools outperforms a large set of near-duplicates; overlap produces inconsistent tool selection across runs. When two tools differ only in a parameter, they should be one tool with a parameter.

2.6.4 How the Orchestrator Composes the Three

The orchestrator is not another model; it is the coded frame around the one agent. Its job is composition, performed once at run start and once per step.

At run start it makes five decisions, in order, none requiring model judgment:

- Route the request to a workflow profile (pattern and classifier work, cheap and logged). The profile names the outcome skill.
- Bootstrap the runtime registry for the tenant: skills, rules, and tool specs from the versioned store (optional tenant overlay). Git is the authoring source of truth; the registry is what runs load. Agents never write live entries.
- Load essential skills: the core discipline skill (always), the profile's outcome skill, and any domain skill the tenant's configuration attaches.
- Attach rules by scope: every rule whose scope matches the profile and stage. Prompt-strength rules are rendered into context; gate-strength rules are registered as checks at their workflow points; code-strength rules are already true and are attached only as citations.
- Build the effective allowlist: the intersection of the outcome skill's tool list with the tenant's enabled sources and the platform registry. Intersection, never union

— a skill cannot grant what the tenant lacks, and a tenant cannot grant what the skill does not permit.

Per step, the composition repeats in miniature. Before each action the orchestrator matches the pending task against the behavioural skills and injects the winning hat's playbook, and may re-route the model tier for that step. When the stage machine advances, the funnel loads that stage's skills and attaches that stage's rules. When the agent proposes a tool call, the orchestrator plays no advisory role at all — the call goes to the executor, which checks registry, allowlist, and scope mechanically.

A compressed trace of the interplay, using the three artifacts above: the request routes to the periodic-report profile → the orchestrator bootstraps the runtime registry, loads `outcome/periodic-performance-report` v3, sees `deterministic_seed: true`, and runs the standing queries before the loop starts, with `rule/query-result-bounds` already shaping every result into artifact-plus-summary. The stage machine opens extraction with exit predicates registered. The loop begins with 24 steps of budget; semantic routing keeps derivation on a standard tier. At step 4 the agent needs period-over-period movement; no special hat is injected (plain data work), and the agent calls `derive_metric` with two artifact references — the executor verifies allowlist and tenant scope, resolves the references, computes, stores provenance. At narrative stage the funnel loads presentation and findings skills; before the prose step the orchestrator injects the communicator's playbook. The draft cites `derive_metric` artifacts for every figure. The consistency gate fails on one mismatch; the loop backtracks, corrects, passes. Presentation uses plan-compose into the host contract rather than freehand HTML. Because the skill header says `review: critic`, the critic's playbook runs before delivery. Total model judgment spent: task reasoning and narrative. Total model judgment spent on governance: none — every boundary was composed, checked, or enforced by code.

The design target is that model capacity is spent on the problem, while composed layers enforce the boundaries.

2.7 Plan-Compose: Judgment Designs, Platform Paints

Named tools for presentation once meant generating HTML in the sandbox. That approach fails in familiar ways: empty chart mounts, invented numbers on retry, host chrome duplicated inside an iframe, and layouts that exist only as model intent and never render. The architectural fix is the same separation the sandbox already taught (intent packaged separately from execution), applied to the deliverable itself.

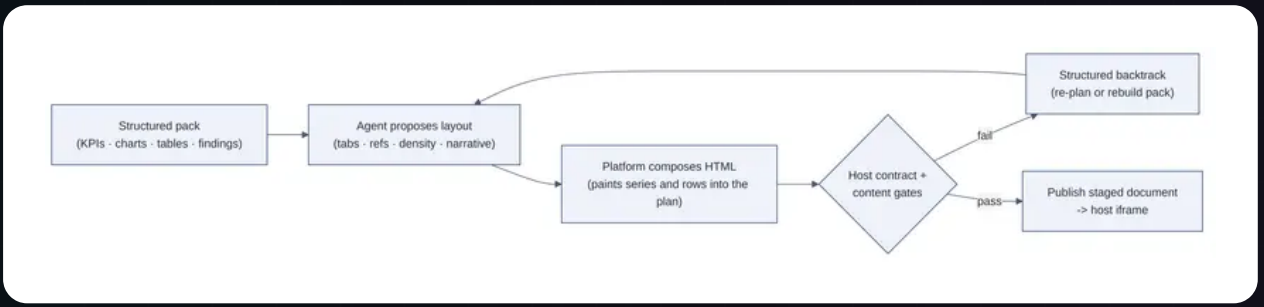


Figure 2.7 — Plan-compose: the agent designs layout, the platform paints evidence

The agent owns judgment: tab order, which visuals sit where, grid density, section summaries, interpretations. The platform owns paint: series into chart mounts, rows into tables, empty visuals skipped and disclosed as gaps rather than hollow placeholders. A host-bridge contract (height reporting, present/print messages, no duplicate chrome inside the document) is enforced as a gate-strength rule at the publish boundary. Content gates reject DOM that looks complete but is not wired to pack data. On compose failure the tool returns a structured backtrack target (re-plan layout, or rebuild the pack) instead of inviting freehand HTML.

Freehand sandbox HTML remains the escape hatch of last resort after structured backtrack fails. The same discipline as Section 3 applies: code is subordinate to named tools. Plan-compose is not a second agent; it is a tool pair on the one loop, with the guardian's checks implemented as gates rather than informal judgment. The pattern generalises beyond reports: whenever the model's creativity is about structure and the reliability problem is about rendering evidence into a fixed host, separate plan from paint.

3. The Sandbox: Computing What No Tool Covers

Named tools cannot anticipate every computation. The escape hatch is a three-step pipeline through which the agent writes and runs its own code, designed so that the flexibility of code arrives without the openness of code.

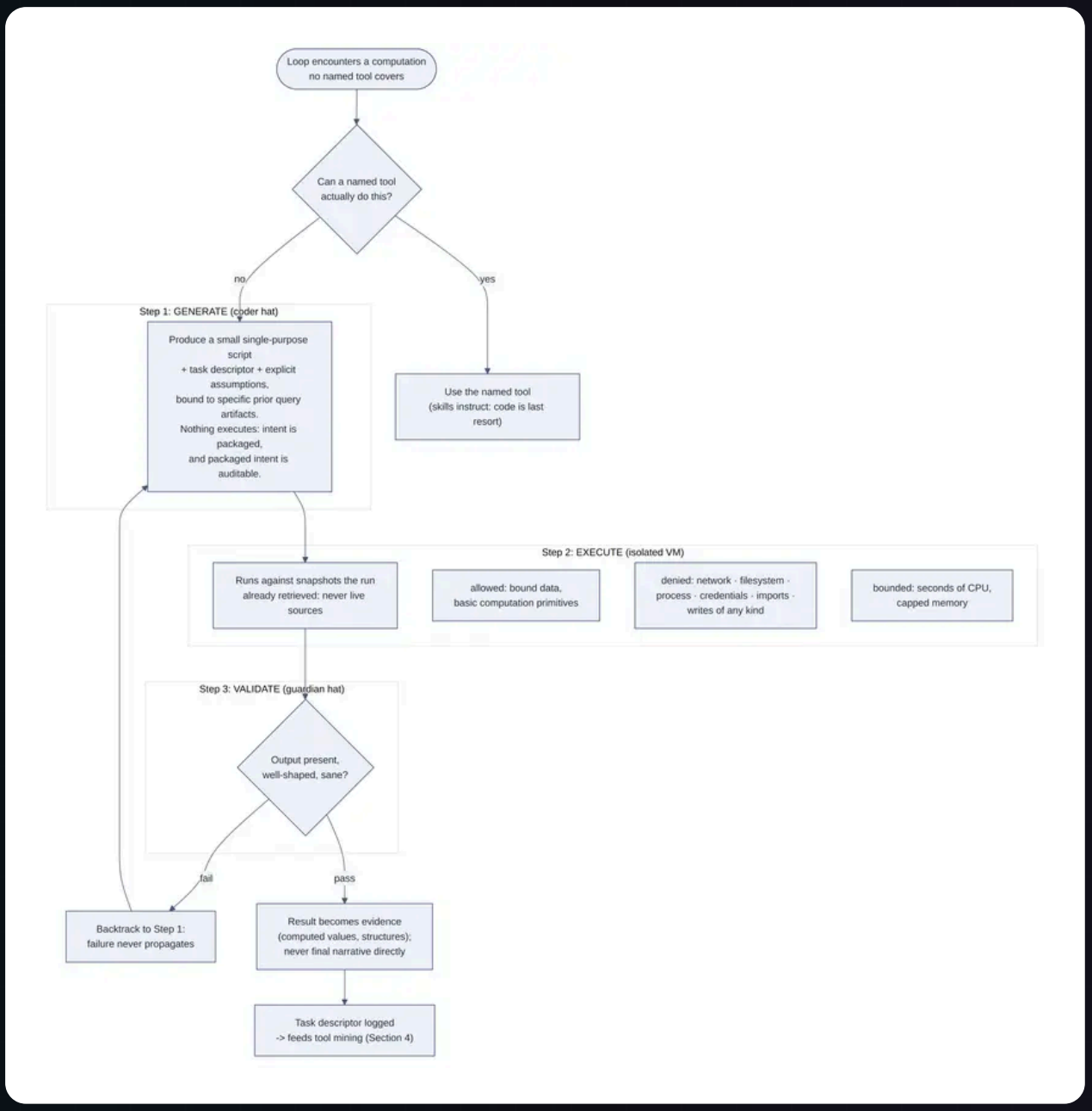


Figure 3.1 — Generate, execute, validate: the sandbox pipeline

Two boundaries keep the sandbox subordinate. Skills instruct the agent to prefer named tools and, for presentation artifacts, plan-compose (§2.7), and to treat freehand code as a last resort. An agent that can write code will otherwise write it constantly. Sandbox output feeds evidence (computed values, composed structures), never final narrative directly; prose still passes through the

communicator and the review gates like every other claim. The sandbox extends what the agent can compute, not what it can assert.

3.1 Worked example: weighted margin bridge

Suppose prior named-tool queries have already produced two artifacts the loop may reference by id:

Artifact	Contents (illustrative)
`q_margin_by_product`	Product-line gross margin for Q1 and Q2
`q_rebate_by_product`	Late rebates booked in Q2, by product

No named derivation tool performs a rebate-weighted reallocation across lines. The coder hat packages intent (still nothing executes):

```python
# assumes: artifacts q_margin_by_product, q_rebate_by_product already in sandb
# bound inputs only; no network / files / imports beyond allowed primitives

margin = load_artifact("q_margin_by_product")   # rows: product, gm_q1, gm_q2
rebate = load_artifact("q_rebate_by_product")   # rows: product, rebate_q2

by_product = []
total_delta = 0.0
for row in margin:
    r = lookup(rebate, product=row["product"]) or {"rebate_q2": 0.0}
    # Q2 reported GM is pre-rebate; adjust for comparability
    gm_q2_adj = row["gm_q2"] - r["rebate_q2"]
    delta = gm_q2_adj - row["gm_q1"]
    by_product.append({
        "product": row["product"],
        "gm_q1": row["gm_q1"],
        "gm_q2_adj": gm_q2_adj,
        "delta": delta,
        "rebate_q2": r["rebate_q2"],
    })
    total_delta += delta
```

```
return {
    "total_delta": total_delta,
    "by_product": sorted(by_product, key=lambda x: x["delta"]),
}
```

The isolated VM runs that script against the two snapshots. A shaped observation returns to the loop; the full table is stored as evidence.

If the output were empty, mistyped, or outside sanity bounds, the guardian path would reject it and force regenerate or an honest gap. Only after validation may the communicator cite `calc_margin_bridge_01` in prose. The task descriptor ("rebate-weighted margin bridge") is what later mining may cluster into a propose-tool candidate (§4).

4. Self-Extension: A System That Grows With Use

The sandbox solves the unanticipated computation once. The harder question is what happens when the same unanticipated need recurs, and this is where the declarative trinity matters. Because skills, rules, and tools are documents and registries rather than hard-coded behaviour, the system can participate in growing them. Three loops run continuously, one per layer of the trinity, but they do not promote with equal automaticity.

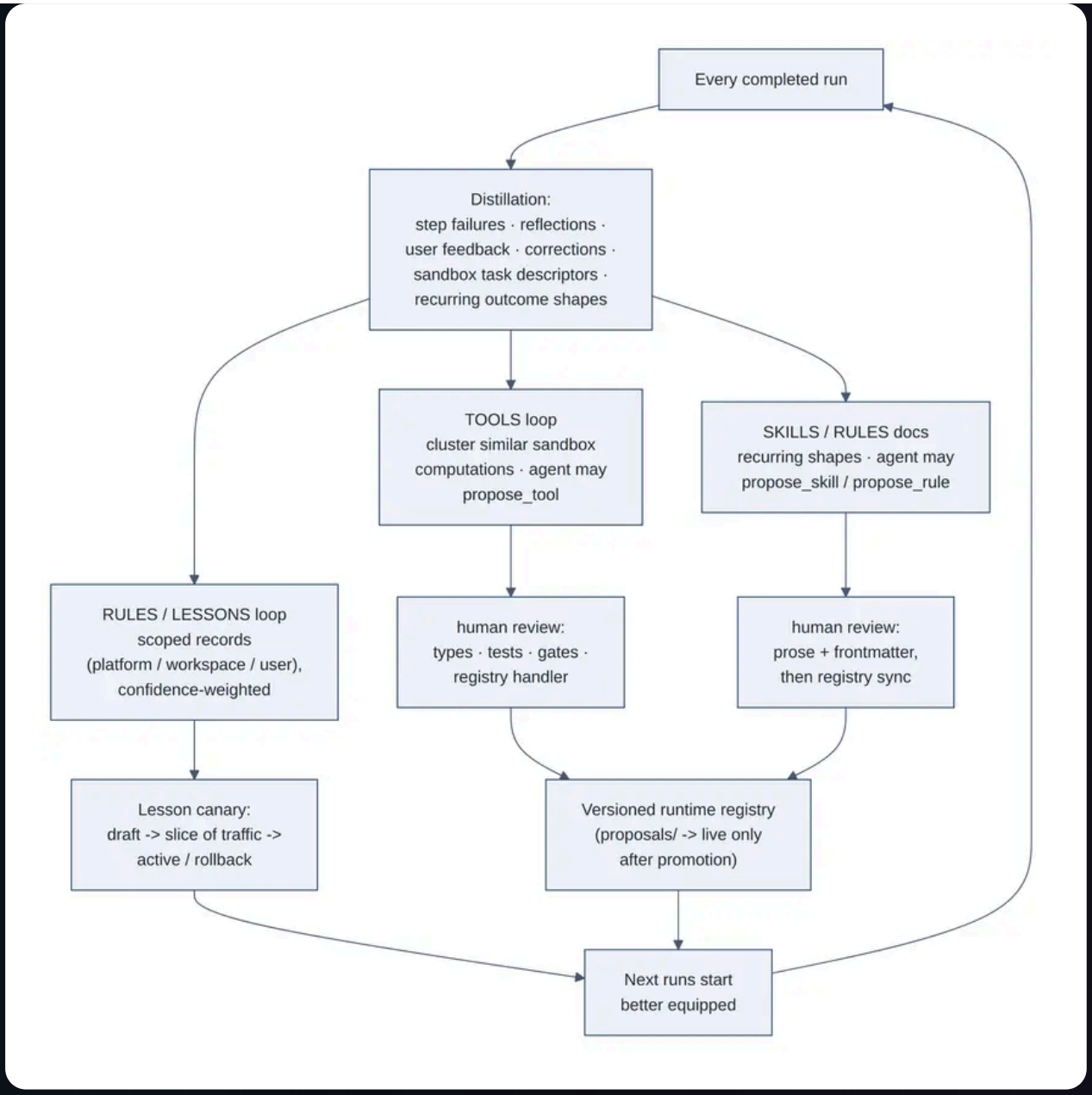


Figure 4.1 — The self-extension loop: distillation, staged promotion, and the versioned runtime registry

New lessons, on the go. After every run, the system distills lessons from what happened: a query pattern that failed against a particular dialect, a formatting choice the user corrected, an approach that thrashed. Each lesson becomes a scoped, confidence-weighted learning record, scoped to the platform when universal, to a workspace or user when preferential, and the top-ranked lessons are injected into future runs' context. Functionally, these are rules the system established for itself from experience. User feedback drives the confidence arithmetic: acceptance strengthens a lesson, rejection weakens or disables it, and an explicit correction becomes a high-confidence lesson verbatim. Lessons follow staged autonomy inside their scope: draft, canary against a slice of traffic, active, or

rollback. The canary evaluates concrete signals — gate failures, retries, acceptance, regression fixtures, whether the lesson ever matches — and expires noise.

New tools and skills, proposed not auto-applied. Every sandbox execution carries its task descriptor; repeated clusters, and standing outcome patterns, are evidence that a named tool or skill deserves to exist. The agent stages candidates through propose-tool, propose-skill, and propose-rule actions into a proposals area of the registry, never into the live catalog. Promotion is human: a proposed tool still needs a typed handler and gates; a proposed skill is a prose review plus frontmatter contract; a proposed gate-strength rule needs the coded check wired at its trust boundary. The trajectory remains deliberate — capabilities are born where flexibility is cheap (sandbox, drafts) and graduate into the versioned registry where they gain types, tests, and guarantees.

The runtime registry as constitution. Skills, rules, and tool specs load from a versioned, encrypted store at run start. The authoring source of truth stays in version control; deploy syncs the registry; deletes are denied so history accumulates as versions. Tenant overlays can specialise without forking the platform library. This is what makes self-extension safe: the model can write a proposal artifact, and humans (or an admin path) promote it; the executor never treats a proposal as live allowlist or playbook.

The architecture still names an evaluation harness so promotion need not wait on traffic alone: replay of recorded runs, a golden set of representative requests, calibrated model-graded scoring for scale, and pooled evidence for platform-scoped candidates, with live canary as the last check for lessons. For skill and tool promotion, human review remains the binding gate; the harness informs the reviewer rather than replacing them. *Note: lesson canaries are the shipped promotion path; the fuller harness for registry promotion is the standard the loops aim at, not every candidate's automatic gate today.*

Self-extension creates an attack surface the loops must contain: feedback is untrusted input. A careless or adversarial user can try to teach the system falsehoods, reject correct answers, "correct" them wrongly, and the distillation pipeline would faithfully turn that into injected rules. Three properties bound the damage. Scoping localises it: user-scoped lessons affect only that user's runs, so a poisoned lesson cannot cross to other users or tenants. Promotion gates it: nothing reaches platform scope, the tool registry, or the skill library without human review.

Confidence arithmetic dampens it: a lesson contradicted by subsequent outcomes loses weight and is disabled rather than argued with. Lessons are also constrained to behavioural adjustments — they shape how the agent works, never what data it may touch; access rules live at strength three, where no distilled lesson can reach.

Human review is required. A system that rewrites its own capabilities and constraints without gates is not autonomous; it is unaccountable, and the trust argument of Section 2 rests on the trinity remaining reviewable. Self-extension under staged promotion captures most of the value: the system can establish new lessons within a day of a failure, and can originate future tools and skills as proposals, without unsupervised self-modification.

5. Persona and Memory: Continuity Without Retraining

An autonomous system that starts every run without memory is not autonomous; it has no continuity. The memory architecture is layered, because different kinds of continuity have different lifetimes, owners, and risks, and collapsing them into one store gets all three wrong.

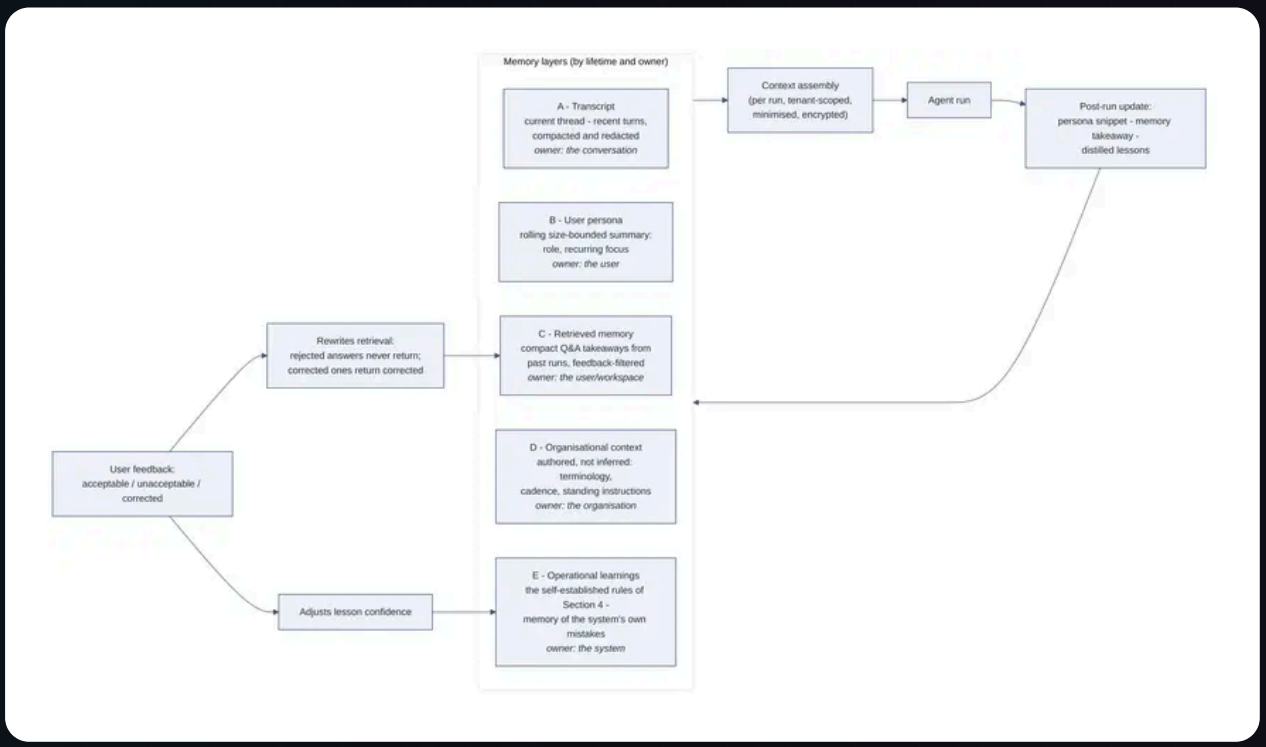


Figure 5.1 — Memory layers by lifetime and owner, and how feedback rewrites retrieval

- **The transcript** carries the current thread: recent turns, compacted and redacted, enough for follow-ups to resolve naturally.
- **The user persona** is a rolling, size-bounded summary per user — their role, their recurring focus — distilled from completed runs a sentence at a time. It tailors emphasis and tone. It is intentionally modest: a compact sketch that is useful when right and harmless when stale, not a dossier.
- **Retrieved memory** stores compact question-and-answer takeaways from past runs and surfaces the most relevant few into new context, filtered through feedback, so rejected answers never return and corrected ones return in their corrected form. Takeaways are kept intentionally brief: memory should carry experience, while analysis is always re-run fresh against current data.
- **Organisational context** is the strongest persona layer, and it is authored, not inferred: the business's description, terminology, reporting cadence, preferences, and standing instructions, maintained by the organisation itself. Inference is a fallback for what nobody wrote down; where the organisation has stated who it is, the system listens rather than guesses.
- **Operational learnings**, the self-established rules of Section 4, are memory of the system's own mistakes, ranked by scope and confidence and injected where relevant.

Every layer is tenant-scoped and minimised: summaries rather than transcripts, identifiers rather than payloads, encryption at rest, and user feedback as a standing correction channel. Memory is both an asset and a liability; the architecture treats it as both.

5.1 Cold Start: Designing for Day One

A memory architecture also has to answer for the tenant that has none of it yet. Three mechanisms are built in, in order of usefulness:

- **Structured elicitation.** Authored organisational context is the strongest layer, so the system makes authoring cheap: onboarding is an interview, not an empty text box. The agent asks for terminology, fiscal calendar, reporting habits, and sensitivities, drafts the organisational context document from the answers, and the organisation confirms or corrects it. The output is the same authored layer as ever; the system merely did the transcription.

- **Starter packs.** Not everything a mature deployment knows is tenant-specific. Domain skills and platform-scoped lessons (query dialect pitfalls, charting conventions, framings an industry expects) carry no tenant data by construction — they live at platform scope precisely because they are universal. A new tenant in a familiar domain inherits this accumulated craft on day one, so the system's self-extension compounds across the platform rather than restarting per tenant.
- **The conservative profile.** For a tenant's first runs, routing applies an intentionally cautious profile: tighter budgets, mandatory review gates, a lower threshold for structured clarification. The system is visibly careful while it is ignorant, and empty memory layers are a signal the router can read. The profile relaxes as lessons, persona, and feedback accumulate.

6. The Autonomous Whole

Assembled, the full system:

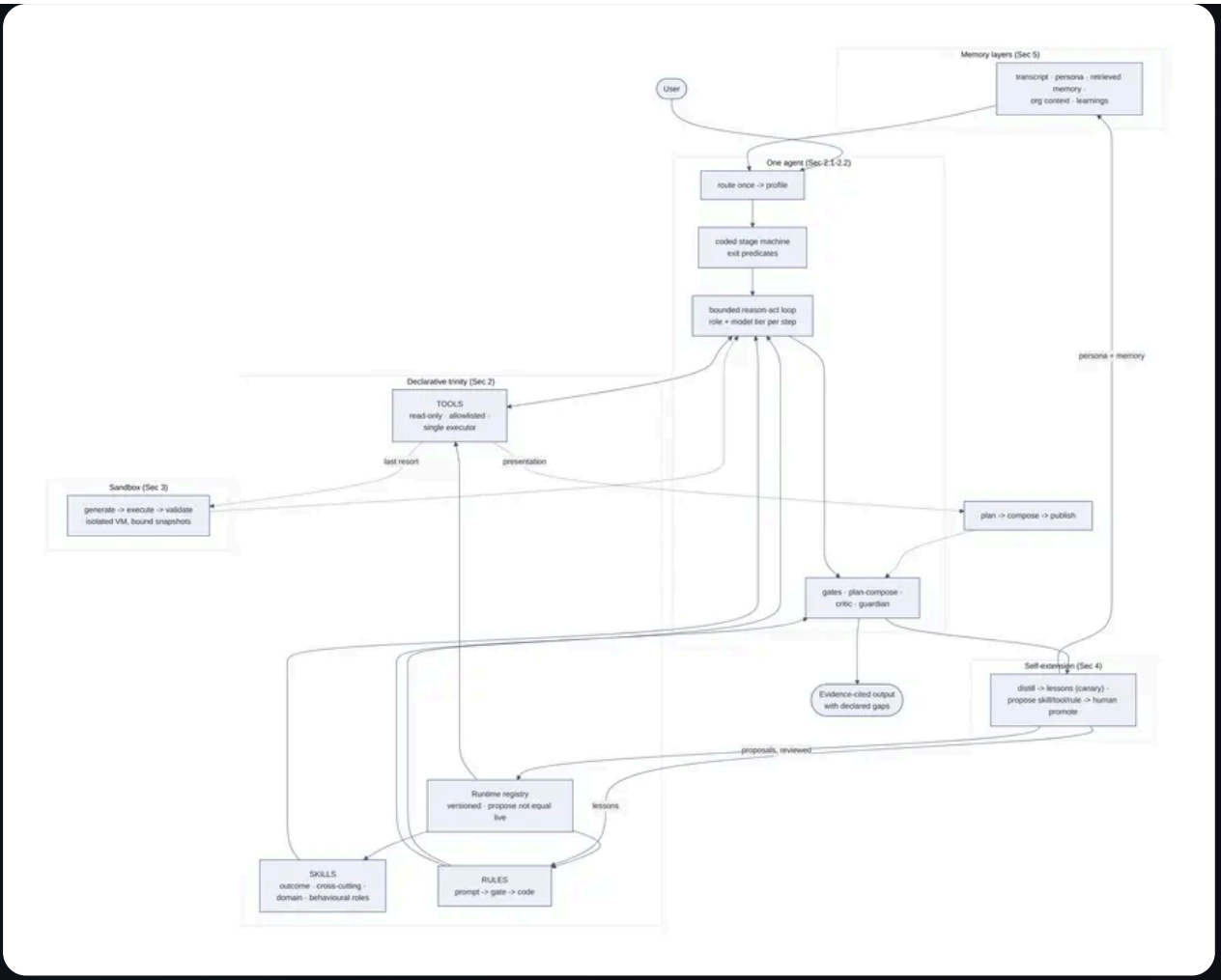
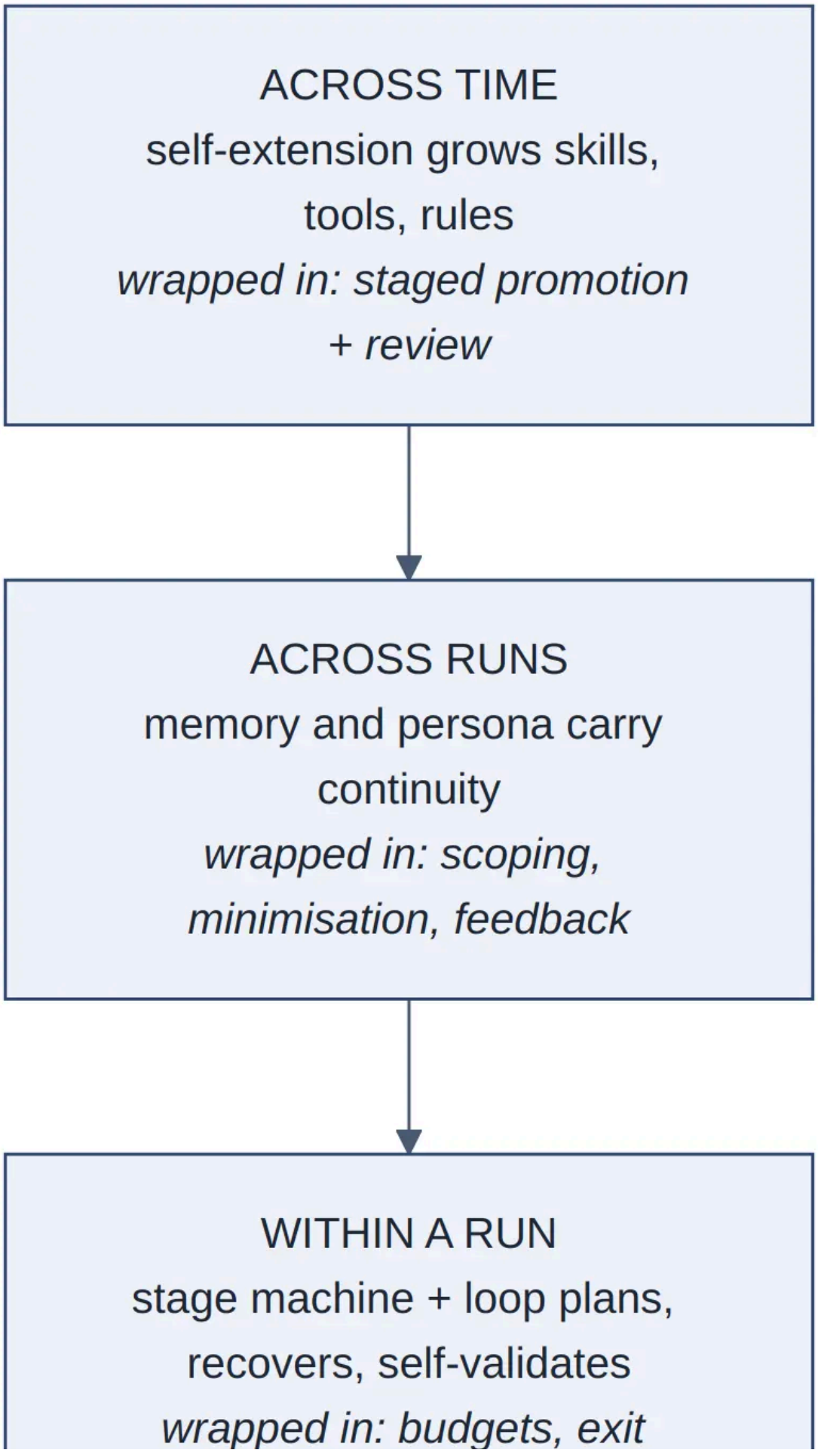


Figure 6.1 — The assembled system: memory, the declarative trinity, the one-agent engine, the sandbox, and self-extension

A run reads left to right: a request arrives and is routed once to a workflow profile; the runtime registry bootstraps; context is composed from the memory layers; skills load by stage funnel and rules attach at their declared strengths; for known-outcome work a coded stage machine orders advancement. The one agent proceeds through its bounded loop, changing hats and model tiers as the work demands — planner to sequence, coder to compute, guardian to check, communicator to synthesise, critic to review — acting only through allowlisted, read-only tools, preferring plan-compose for presentation, computing the irregular in the sandbox, recovering on the same timeline, and passing gates it cannot bypass. The output carries its evidence and its declared gaps. After the run, the system turns the experience into scoped lessons, stages capability proposals for human promotion, and updates its picture of the user, so the next run starts slightly better equipped than this one did.

Autonomy, in this architecture, is not a single property but a stack of them:


ACROSS TIME
self-extension grows skills, tools, rules
wrapped in: staged promotion + review
ACROSS RUNS
memory and persona carry continuity
wrapped in: scoping, minimisation, feedback
WITHIN A RUN
stage machine + loop plans, recovers, self-validates
wrapped in: budgets, exit

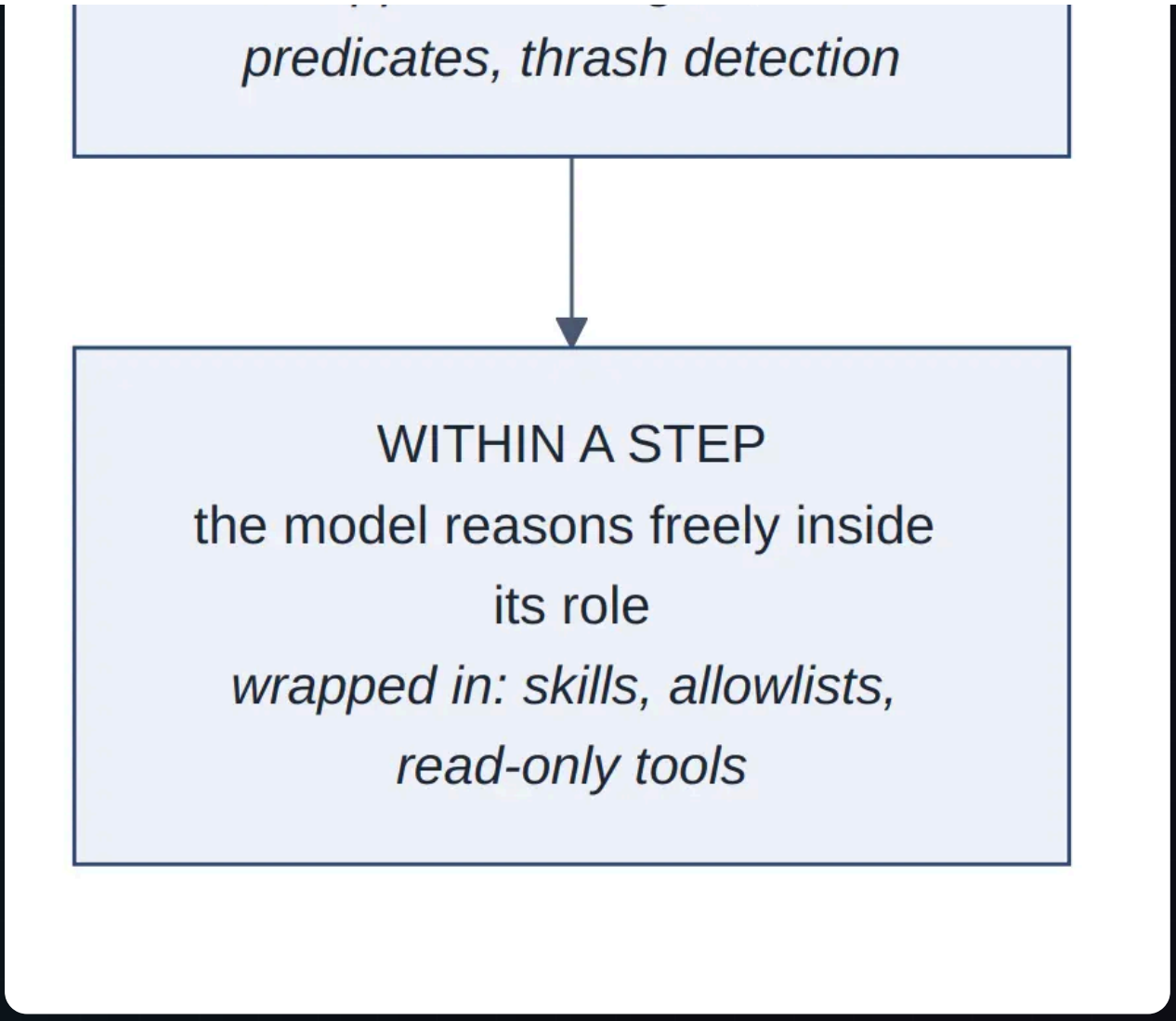


Figure 6.2 — Autonomy as a nested stack, each level wrapped in the constraints of the level below

Each level depends on the constraints of the levels below it. That nesting is what makes autonomy tractable.

6.1 A Run, End to End

Two traces below use illustrative figures. Map the domain to your own setting; the machinery is what transfers.

6.1.1 Ad-hoc analysis (question → cited answer)

Request: "Why did gross margin drop last quarter?"

Route. Ad-hoc analysis profile: modest tool budget, prose output, no formal critic pass required. Context: finance persona, preference for brevity; fiscal calendar from organisational context; one operational learning that margin lands pre-rebate.

Plan (planner hat). Confirm the drop; decompose by product; check one-offs and rebate timing.

Act. Named-tool queries return bounded summaries and artifact ids. Illustrative evidence after extraction:

Product	GM Q1	GM Q2 (reported)	Rebate Q2	GM Q2 (adj.)	Δ vs Q1
Widgets	28.0%	26.4%	1.5pp	24.9%	-3.1pp
Gadgets	31.2%	30.8%	0.2pp	30.6%	-0.6pp
Other	22.0%	21.5%	0.0pp	21.5%	-0.5pp
Total	**27.1%**	**26.0%**	–	**25.0%**	**-2.1pp**

Reported total Δ (–1.1pp) understates the movement once rebates are applied. No named tool performs that reallocation, so the sandbox path in §3.1 runs and yields `calc_margin_bridge_01` (total Δ –2.1pp after adjustment; Widgets –3.1pp).

Deliver (communicator hat). Draft cites artifacts, not invented arithmetic:

> *Gross margin fell about 2.1 percentage points quarter-on-quarter after rebate adjustment (`calc_margin_bridge_01`). Widgets account for most of the drag (–3.1pp). Reported Q2 margin is pre-rebate; treating it as final would understate the decline.*

A consistency gate checks every figure in that prose against artifacts. One source is three days stale; the answer discloses it rather than hiding it.

Distill. The user marks the answer acceptable (rebate lesson confidence rises). The sandbox task descriptor joins a cluster that may later become a propose-tool draft.

6.1.2 Known-outcome pack (stage machine → plan-compose → backtrack)

Request: "Build this month's management pack for the leadership review."

Route. Periodic-report profile: deterministic seed on, stage machine on, critic review required, presentation via plan-compose.

Stages (coded exits, same agent). Connector bootstrap passes (`SELECT 1`). Schema discovery and query plan complete for every required section, or a gap is documented. Extraction and derivation fill the pack JSON. Narrative findings must cite query/derive artifacts before assembly advances. Structural assembly and the quality gate block a hollow pack. If quality fails within budget, recovery stays on the main loop under pinned run scope (same transcript, escalating strategy), not a sub-agent.

Presentation (plan-compose). After a valid pack, the agent proposes layout (no freehand HTML):

```JSON
{
  "tabs": [
    {
      "id": "exec",
      "title": "Executive summary",
      "kpiRefs": ["gm_pct", "revenue"],
      "findingRefs": ["f_margin_drag"]
    },
    {
      "id": "margin",
      "title": "Margin bridge",
      "chartRefs": ["ch_margin_by_product"],
      "tableRefs": ["tbl_margin_bridge"],
      "summary": "Widgets drive the post-rebate decline."
    },
    {
      "id": "gaps",
      "title": "Data gaps",
      "gapRefs": ["gap_rebate_lag"]
    }
  ],
```

```
  "gridColumns": 4
}
```

Compose paints series and rows from the pack into that plan. First attempt fails the host-contract / content gate: `ch_margin_by_product` is referenced but the pack chart has an empty series. The tool returns a structured backtrack to `propose_pack_html_layout` (or rebuild the pack), not a prompt to invent substitute series. The agent removes the empty chart, discloses it on Data gaps, and republishes. Verification then reconciles KPI strings in the HTML against pack values before delivery.

Distill. Gate failure and empty-series skip become candidate lesson material for the next pack run; no live skill is rewritten without human promotion.

The hats change across both traces; the agent does not multiply.

7. Threat Model and Residual Limits

A paper that recommends granting autonomy should account for how the architecture fails, both under attack and under ordinary conditions it does not suit.

7.1 Threat Model

The layered design exists to hold specific threats. Where each is held:

Threat	Vector	Held by
Prompt injection	Instructions embedded in retrieved documents or query results ("ignore previous instructions...")	Strength-3 boundaries: an injected instruction can change what the model says, but the executor still rejects any tool call outside the registry, allowlist, or tenant scope, and no exfiltration-capable tool exists to invoke
Data exfiltration	Model induced to transmit data outward	Read-only surface by construction: no network, write, or transmission tool

Threat	Vector	Held by
		exists; sandbox denies network and filesystem; output size is bounded
Cross-tenant leakage	Confused-deputy access to another tenant's data	Tenant scoping enforced at the executor and the infrastructure layer, not the prompt; memory layers are tenant-scoped at rest
Memory poisoning	Adversarial feedback distilled into rules (Section 4)	Scoping, human-reviewed promotion, confidence decay; lessons are behavioural only and cannot alter access
Sandbox abuse	Generated code probing for escape or resources	Isolation (no network, filesystem, process, imports, credentials), CPU and memory caps, execution only against already-retrieved snapshots: a successful escape yields data the run was already entitled to read
Capability creep	Self-extension gradually widening the action surface	Propose tools stage drafts only; live skills/rules/tools enter the runtime registry only through human promotion; lesson canaries stay behavioural and scoped
Hollow presentation	Model ships DOM that looks finished but charts/tables do not paint or invent numbers	Plan-compose (§2.7): platform paints pack data; host-contract and content gates; structured backtrack; freehand HTML last resort

The pattern across the table restates the thesis: every threat that matters is held at strength three, or at a gate the model cannot bypass, where the model's output is not consulted as authority.

7.2 Residual Limits

Earlier drafts of this paper listed limits that later drafts claimed were "now in the body." Production required a more careful ledger. Mechanisms that did ship — the coded stage machine and artifact-referenced compaction (§2.1), semantic model routing (§2.2), plan-compose (§2.7), lesson canaries and propose-then-promote

(§4), cold-start machinery (§5.1) — live where they belong. What remains below is the remaining limits, including claims that are still aspirational rather than shipped.

- **Decision-making stays serial by design.** Critic and guardian passes are hats and tools on one timeline; evaluation fan-out (§2.2) remains an optional latency optimisation, not a shipped escape from seriality. A workload whose decisions must parallelise (independent agents pursuing independent goals) is outside this architecture's boundary, and should be built as separate systems with contractual interfaces, each internally composed this way.
- **Stages gate work; context is still shared.** The stage machine (§2.1) is real; fresh-context phases are not. Compaction and artifact ids buy most of the ceiling benefit, but hunches formed early can still drown in a long transcript, and a continuation checkpoint that reloads the same message array is not a clean phase handoff. Schemas that carry open questions and suspicions, not just conclusions, remain the direction.
- **Cold start is softened, not solved.** Elicitation, starter packs, and the conservative profile (§5.1) make day one competent, but tenant-specific judgment (this organisation's tolerance for speculation, this user's actual priorities) only accumulates through use. The first weeks are still the system's weakest; the architecture's contribution is that the system behaves like it knows that.
- **Promotion evidence is uneven.** Lesson canaries are live; the fuller evaluation harness (replay, golden set, calibrated judge) for registry promotion informs reviewers more than it auto-promotes. Coverage still inherits fixtures: a candidate can pass every recorded case and fail on the request nobody recorded.
- **The multi-agent critique remains experiential.** The architecture now defines metrics that would settle it (cost per completed outcome, error-attribution time, regression isolation), and the unified compose loop plus stage machine strengthen the case, but this paper still reports one production setting, not a controlled comparison. Readers with contested architecture decisions should run the comparison on their own task suite rather than take either side's word.
- **Structure rations judgment; it does not create it.** Deterministic seeds, derivation tools, plan-compose, and gates shrink the surface where model quality matters; semantic routing (§2.2) sends frontier models to the steps that need them; the regression suite gates model swaps; gate-failure rates surface

drift early. But the residue is irreducible: on the steps where only judgment will do, the system is exactly as good as the model wearing the hat. The architecture bounds the damage of a weak model; it cannot supply the insight of a strong one.

8. Practitioner Checklist

The trinity

- Is behaviour composed from declarative, versioned layers (playbooks, guardrails, action registries) rather than hard-coded or prompt-monolithic?
- Are roles stances one agent adopts per step, with a single audit trail, or separate agents whose coordination nobody can reconstruct?
- Do skills load progressively by stage, keeping context proportional to the work at hand?
- Does every rule declare its enforcement strength, with critical constraints held by gates and infrastructure rather than instruction?
- Is the tool surface read-only by construction, allowlisted per skill, tenant-scoped, and sanitised, enforced at the executor rather than the prompt?
- Do decisions stay on one serial, auditable timeline? (If evaluation is parallelised, does it only score and never act?)
- For known-outcome work, does a coded stage machine block advancement until exit predicates hold, while context stays one compacted loop with artifact handoffs and occupancy metrics?
- Does semantic routing bind light / standard / frontier models per step, with the model treated as a versioned dependency gated by a regression suite?
- Prefer plan-compose for hosted presentation: agent designs layout, platform paints evidence, host-contract gates reject hollow DOM, backtrack before freehand HTML?

The sandbox

- Does generated code execute only against bounded snapshots the run already holds — no network, no filesystem, no credentials, no live sources?
- Is generation separated from execution, and execution from validation, so intent is auditable and failure backtracks instead of propagating?
- Is code a governed last resort behind named tools, and its output evidence rather than final narrative?

Self-extension

- Do failures, corrections, and feedback distill into scoped, confidence-weighted lessons that reach future runs via canary?
- Can the agent propose skills, rules, and tools into a proposals area (never the live registry), with human promotion required?
- Is the runtime registry versioned and bootstrapped at run start, with authoring in version control and sync on deploy?
- Is there an evaluation harness path (replay, golden set, judge) so promotion need not wait on traffic alone, with clarity about what auto-promotes today versus what still needs a human?

Persona and memory

- Is memory layered by lifetime and owner — thread, user, organisation, system — rather than one undifferentiated store?
- Does feedback rewrite retrieval, so the user's corrections become the system's memory?
- Is authored organisational context preferred over inference, and every layer minimised, scoped, and encrypted?
- Does day one work: elicited context instead of empty forms, platform-scoped starter craft, and a conservative profile that relaxes as memory accumulates?

Threats

- If an injected instruction fully controlled the model's output, what is the worst action the executor would still permit? (If the answer is anything beyond a wrong answer, a boundary is missing.)
- Is untrusted feedback prevented from becoming platform-wide behaviour without review, and are distilled lessons behavioural only (never access-widening)?
- Would a sandbox escape yield anything beyond data the run already held?
- Can a presentation ship that looks finished but does not paint pack data, or invent numbers on retry, without failing a host-contract or content gate?

9. Conclusion

The earlier paper argued that the quality of an answer depends on the process that produces it. This paper extends the argument to the system itself: the trustworthiness of an autonomous agent depends on how its behaviour is constituted, and the constitution that survived production is a declarative one. One agent, whose many roles are skills it wears rather than agents it consults. Guardrails that declare their own strength, so the critical ones hold structurally while the stylistic ones stay cheap to change. An action surface that is read-only, allowlisted, and audited at a single point. A coded stage machine that the model cannot skip. Plan-compose so judgment designs and the platform paints. A sandbox that grants the power of code without the openness of code. A growth loop through which the system's own experience — its failures, its repetitions, its users' corrections — becomes new lessons immediately, and new tools and skills only as reviewed proposals into a versioned registry.

What made this tractable was not model capability alone; models were capable throughout. The decisive choice was to make every layer of the system's constitution readable by humans, who must review it, and by the system itself, which must be bound by it and can therefore help write it. An agent system built this way can be given operational autonomy because each level of autonomy is constrained by structure that holds when the model errs. The engineering problem of agent systems is therefore not enabling models to act, but building systems that remain accountable when they do.

The patterns transfer to other domains where agents act on consequential data; adaptation and critique from practitioners are welcome.

How to cite APA-style BibTeX

```
Sandeep Kavety (2026). One Agent, Many Hats: Rules, Skills, and
Tools as the Architecture of an Autonomous, Self-Extending Agent
System (Version 1) [Working paper].
https://sandeepkavety.com/writing/one-agent-many-hats
```

Copy to clipboard

PREVIOUSLY IN THE SERIES

Conversational Decision Intelligence

Flow, Reasoning, and Continuous Learning in Natural Language Systems over Structured and Unstructured Data