PREFRONTAL: The Recipe Execution Substrate — How a Composable, Self-Authoring, Looping Workflow Engine Becomes an Agent's Executive Function

Oscar Serra, Jarvis · Independent Research

19 June 2026 — v4.8

Abstract

The dorsolateral prefrontal cortex does not *do* tasks. It holds the plan, selects the right behavioural program for the moment, sequences sub-routines, suppresses the impulsive response, watches whether the program is working, and re-allocates effort when it is not. It is the brain's executive: not a worker, but the structure that makes work coherent. We argue that a deployed language agent needs the same thing, and that the right shape for it is neither a guard rail nor a manager-agent but a **recipe execution substrate** — a layer that turns each turn into an executive-function cognitive cycle of *plan* → *match/compose/author* → *execute-with-loops* → *observe* → *adapt-effort*.

A recipe is a structured workflow: ordered steps, tool hints, success gates, and failure handlers, written as a portable document rather than as code. The substrate does six things with recipes that together make them an execution engine rather than a static library. (1) It **matches** the user's intent to recipes with a fuzzy, stemmed, edit-distance-tolerant scorer that reports a confidence tier — and that admits negative as well as positive evidence, so a recipe can declare the prompts for which it must *not* be chosen — and seeds a plan automatically at the start of every turn. (2) It **composes** recipes from recipes — a recipe may merge another recipe's steps into its own at plan-build time, or call a sub-recipe at runtime, both cycle- and depth-guarded, exactly as functions call functions. (3) When nothing matches, it **authors** a new recipe on the fly, validates it, persists it, and makes it matchable on the very next turn — closing the loop on its own coverage gaps. (4) It supports bounded **loops** over steps (run N times / until-dry / until a marker appears), the element that lifts recipes from straight-line checklists to a structured execution substrate and closes the long-standing structural gap against hand-coded agent workflows. (5) It **routes effort dynamically**, classifying each turn as trivial / standard / deep / ultra and using that classification to *drive* the turn — model tier, thinking budget, orchestration mode, and token generosity all scale to the work. (6) It makes all of this **observable** through a dedicated RECIPES panel that renders the provenance trail (searched → matched → merged → composed → authored), per-subagent vitals, and the compact-versus-expanded decision trail, while keeping orchestration mechanics out of the substantive chat.

The thesis is that intelligence in a deployed agent is not primarily a property of the model. It is a property of the *structure the model executes inside* — and that structure is most powerful when it is composable, self-authoring, looping, effort-adaptive, and observable. This paper specifies that substrate, grounds the brain metaphor in concrete mechanism, gives language-agnostic pseudocode for each of the six capabilities, and reports which of the substrate's harder extensions — durable resume, a self-improving library, external acquisition, and a versioned marketplace — are implemented and running, and which one — searched execution — remains specified but unbuilt. Throughout, we position the substrate against the fresh open-source ecosystem it shares a problem space with — the headroom project's reversible context-compression and the agent-skills plugin's anti-trigger and load-constraint discipline — and fold their stronger ideas into the substrate's own seams where they sharpen a claim.

Keywords: recipe execution, executive function, agent orchestration, workflow composition, self-authoring workflows, bounded loops, effort routing, observability, cognitive cycle

1. Introduction — From Library to Substrate

1.1 The prefrontal analogy, taken seriously

When a person sits down to debug a stubborn problem, very little of the work is the "debugging." Most of it is executive: deciding *that* this is a debugging task and not a redesign task, recalling the program ("reproduce, then isolate, then fix, then verify"), holding that program in working memory while the hands and eyes do the reading and editing, suppressing the urge to patch the first symptom, noticing when the current step has stopped producing new information, and spending more attention when the problem is hard and less when it is trivial. The prefrontal cortex is the organ of that executive layer: working memory, task-set selection, sequencing, response inhibition, and effort regulation (Miller & Cohen, 2001).

A language model is an extraordinary worker and a poor executive. Turn to turn it can read, reason, write, and call tools at a level that dwarfs human working memory for a single step — and yet, left to itself, it repeats the same structural mistakes: it edits before it reproduces, it codes before it designs, it searches before it scopes, it declares victory before it verifies. The deficit is not capability. It is the absence of an executive layer that holds and sequences the *program*.

This paper describes that layer. We call it PREFRONTAL, and we argue its correct form is a **recipe execution substrate**: a thin layer wrapped around the agent that, on every turn, recognises the task, selects or assembles the appropriate behavioural program, runs it with the agent as the worker, watches the execution, and regulates how much effort the turn deserves. The brain metaphor is not decoration; each capability below maps onto a specific executive function, and we name the mapping explicitly as we go.

1.2 What a recipe is, and why it is the right unit

A **recipe** is a structured workflow expressed as a portable document. It carries metadata (what intents it serves, what it is for), a set of triggers, an ordered list of steps — each with a short instruction, optional tool hints, and a *done-when* success criterion — and, where relevant, constraints, safety notes, and a record of the failure modes it was designed to prevent. A recipe is *not* a prompt and *not* a script.

- A **prompt** says "fix this bug." It is a wish.
- A **script** says, in code, "run command X, parse output, branch on line 12." It executes mechanically and shatters on the first input it did not anticipate.
- A **recipe** says "reproduce the failure with evidence; *then* trace the root cause to a file and line; *then* apply the smallest fix; *then* verify the original error is gone." It is structure that an intelligent worker can follow, adapt, or — with justification — override.

The recipe is the right granularity because it is the unit at which operational knowledge actually accumulates. A team that has debugged a thousand issues does not encode that experience as a better model; it encodes it as the discipline of *reproduce before you fix*. A recipe is where that discipline lives in a form that is human-readable, versionable, composable, and — as we will show — machine-authorable.

1.3 The gap this paper closes

A recipe *library* is useful but inert: a shelf of playbooks the agent consults when it remembers to. The contribution of this work is to make the library a **substrate** — an active engine with six properties that, taken together, are what separate a checklist from a programming language for behaviour:

1. **Matching** turns "the agent should pick the right playbook" into "the right plan is already seeded before the agent's first token," via a fuzzy scorer with confidence tiers that admits both positive evidence (tag/title/summary hits) and negative evidence (a recipe's own anti-triggers). (Task-set selection.)
2. **Composition** lets recipes be built from recipes, both statically (step-merge) and dynamically (runtime sub-recipe calls), so workflows reuse workflows instead of duplicating them. (Hierarchical program structure.)
3. **On-the-fly authoring** lets the substrate write a new recipe when none fits, validate it, and add it to the library mid-conversation — so coverage gaps heal themselves. (Skill acquisition.)
4. **Loops** let a step repeat under a bounded condition, the single feature that elevates recipes from straight-line sequences to a substrate expressive enough to encode iterative work. (Sustained, monitored repetition.)
5. **Effort routing** classifies the turn and *drives* it — choosing model tier, thinking budget, orchestration mode, and token generosity to fit. (Effort regulation.)
6. **Observability** surfaces the whole life-cycle in a dedicated panel, so a human can see what program is running, where it is, and how each worker is doing, without that machinery polluting the substantive answer. (Metacognitive monitoring.)

The rest of the paper specifies each in turn (Sections 3–8), after establishing the architecture and the economics that make it affordable (Section 2). Section 9 integrates the substrate with the surrounding cognitive systems; Section 10 gives evaluation design; Section 11 reports which harder extensions beyond the six core capabilities are implemented and which one remains open; Section 12 concludes; Section 13 states the surrounding cognitive stack the substrate depends on.

2. Architecture — The Executive-Function Cycle

2.1 The cycle

PREFRONTAL wraps every interaction in a five-phase cycle that mirrors executive function:

A defining design choice: the EXECUTE phase uses *the same agent* as the worker. PREFRONTAL does not introduce a separate "manager agent" with its own personality and its own context. It seeds structure — a plan — and lets the worker run inside it. This preserves the property that makes language agents valuable (open-ended competence) while adding the property that makes them reliable (a default path of thoroughness). The recipe is a track, not a cage: the worker can deviate from a step when it can justify the deviation, because the gates are evaluated by judgement, not by a state machine.

When the EXECUTE phase fans a step out across multiple subagents, the substrate treats each branch as a settled result rather than a value-or-silence: a branch that throws surfaces a typed, index-attributable failure rather than vanishing as a null, the result order and the await-all barrier are preserved, and the step can settle its survivors and report a partial outcome instead of failing wholesale. Parallel execution under the substrate is therefore partial-failure-tolerant and attributable, not all-or-nothing-and-silent — a reliability property a real execution engine must have, and the precondition for the per-subagent vitals the panel renders (§8.3).

2.2 Where the structure comes from, and when it is cheap

The substrate is engineered so that the executive overhead is near-free for the common case and is spent only where it pays. Three mechanisms make this true.

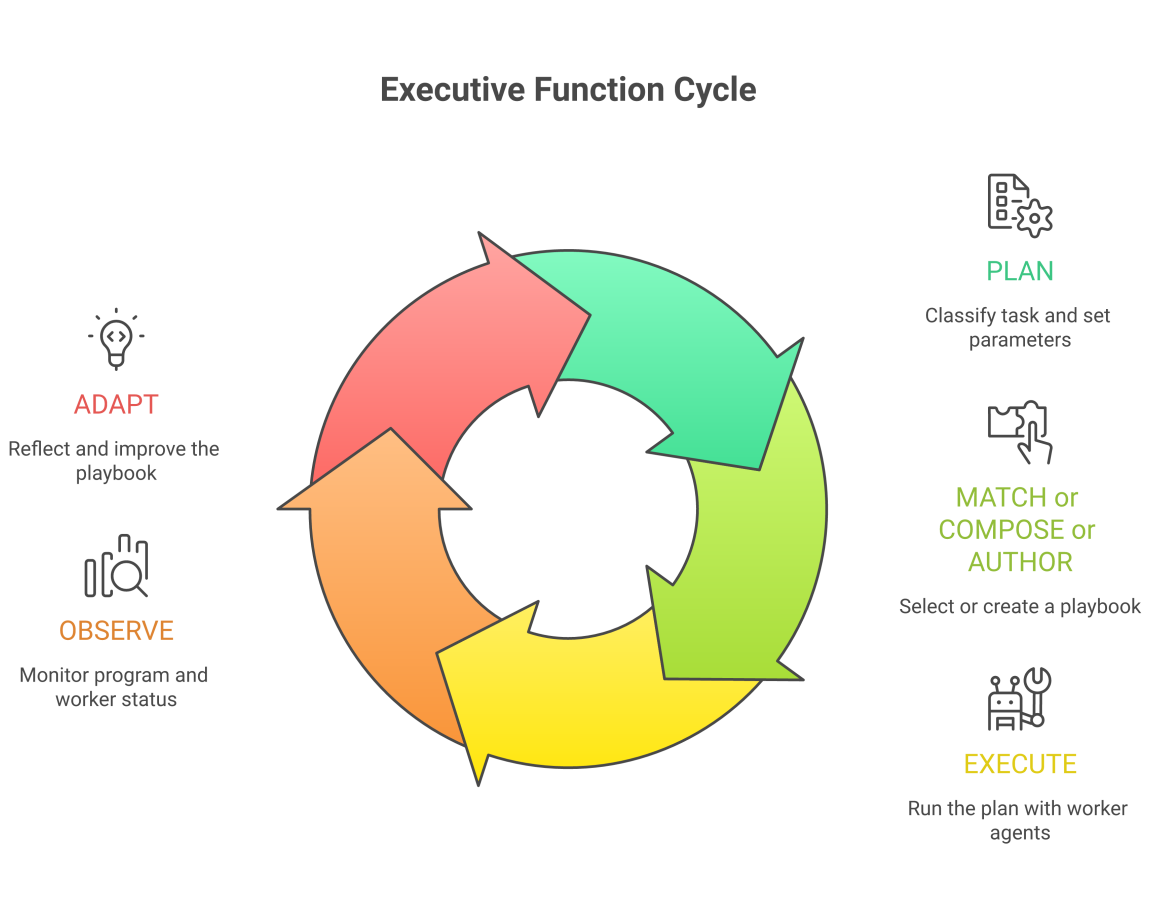


Figure 1. The five-phase executive cycle, annotated with the executive function each phase implements. The dashed back-edge shows recipe improvement (the ADAPT phase) feeding the next turn.

First, **matching is local and lazy.** Task-set selection runs at the very start of every turn against an in-memory index of recipe metadata, cached and invalidated by the recipe directory's modification time. There is no network call and no model call on this hot path; the scorer is a few hundred microseconds of token arithmetic. A plan is therefore seeded *before* the worker's first token, at negligible cost.

Second, **effort routing gates depth.** A trivial turn ("thanks," "what time is it") is classified as trivial and the substrate stays out of the way entirely — no plan injection, no extra reasoning, minimal tokens. The machinery of the cycle is fully engaged only when the turn warrants it. This is the analogue of the cortex spending metabolic budget proportional to task difficulty rather than firing at full intensity for every stimulus.

Third, **follow-up cognition reuses context.** When a phase does require a model call (a reflection pass, or a tiebreak judge when two recipes score within a hair of each other), that call shares almost the entire context of the main turn — the same identity, the same retrieved memory, the same conversation. A follow-up call that differs only in its instruction is therefore far cheaper than a fresh one. The executive layer is cheap precisely where it is invoked most.

2.3 Plans as the runtime trace of recipes

A recipe is a template; a **plan** is the instance of one recipe (or several, merged) executing inside a specific session. The plan is a persisted, structured document: it records which recipe(s) seeded it, the ordered steps, and the live status of each step (`pending` → `in_progress` → `done` | `error`). A single invariant governs it: *at most one step is* `in_progress` *at a time per plan*, enforced by the store rather than by the caller, so the executive's working-memory "cursor" is always unambiguous.

Because the plan is persisted and seeded automatically at turn start, the substrate gains crash-resilience: if the process restarts mid-task, there is an `in_progress` plan on disk describing exactly where the work was, and the executive resumes from it rather than re-deriving intent from scratch. The plan schema reserves a per-step `artifact` field and a plan-level `currentStep` cursor for exactly this purpose; Section 11.1 reports the resume path that reads them back, and how each carried-forward artifact is made a *reversible* pointer rather than a lossy truncation. The plan *is* the agent's working memory, externalised — and §11.1 is what makes that claim lossless rather than approximate.

3. Capability 1 — Recipe Matching (Task-Set Selection)

3.1 The problem with literal matching

The naïve approach to recipe selection is keyword matching: if the message contains "bug," select the debug recipe. It is brittle in exactly the way human task recognition is not. A person reading "the build keeps dying after I touched the config" instantly recognises a debugging situation; a literal matcher sees no shared token with a recipe tagged `bug, error, crash` and selects nothing. Worse, a silent non-selection is invisible: the agent proceeds with no program, and no one knows the executive failed to engage.

3.2 Fuzzy, stemmed, edit-distance scoring

The matcher tokenises the prompt (lower-casing, dropping stop-words and very short tokens) and scores each recipe's metadata against those tokens with a graduated notion of "match" that tolerates the ways natural language varies:

- **Exact equality** ("debug" == "debug").

- **Shared stem** after stripping common English inflections, so *debugging*, *debugged*, *debugger* all reduce to the same root as *debug*.
- **Prefix overlap** for tokens of reasonable length, so *configuration* matches *config*.
- **Edit-distance-1** on the stemmed forms for longer tokens, absorbing typos and minor morphological drift.

The graduation matters: tiny tokens are matched only by equality (to avoid spurious hits), while longer tokens earn the looser rules. Scoring is weighted by *where* the match lands: a hit against a recipe's hand-curated trigger tags counts most, a hit in the title less, a hit in the summary least. Multi-word tags match as phrases; single-word tags and title/summary words match fuzzily.

```
function SCORE_RECIPE(prompt_tokens, recipe):
    score <- 0
    for tag in recipe.tags:
        if tag is multi-word:
            if prompt contains tag as a phrase: score += 5
        else if any prompt token equals tag:           score += 3
        else if any prompt token FUZZY-MATCHES tag:     score += 2
    for word in tokens(recipe.title):
        if any prompt token equals word:                score += 2
        else if any prompt token FUZZY-MATCHES word:    score += 1
    for word in tokens(recipe.summary):
        if any prompt token FUZZY-MATCHES word:         score += 1
    for anti in recipe.not_when:                        # negative evidence
        if anti is multi-word and prompt contains anti as a phrase: score -= 5
        else if any prompt token FUZZY-MATCHES anti:                score -= 3
    return score

function FUZZY-MATCH(a, b):
    return a == b
        or (stem(a) == stem(b) and len >= 3)
        or (len(a),len(b) >= 4 and one is a prefix of the other)
        or (len(a),len(b) >= 5 and edit_distance(stem(a), stem(b)) <= 1)
```

3.3 Negative evidence — anti-triggers

A purely *additive* matcher has a real expressive gap: it can score a recipe upward on tag, title, and summary overlap, but it has no way for a recipe to declare *the prompts for which it must not be chosen*. The consequence is concrete: a recipe that fuzzily shares tokens with an out-of-scope prompt can win on lexical score alone, and the only correction available is the low-confidence advisory (§3.3 confidence tiers, below) — a soft signal, after the fact. The discipline that closes this gap is borrowed from the agent-skills plugin (Osmani, 2026), where every skill carries an explicit "When NOT to use" anti-trigger section that is as load-bearing as its triggers.

The substrate adopts the same idea as a first-class matcher input: a recipe may declare a `not-when:` set — anti-triggers expressed exactly like tags — and a fuzzy hit against an anti-trigger *subtracts* from the score (above), with a strong-enough negative hit able to veto a tier promotion outright (§3.4). This makes task-set selection bidirectional: a recipe is now defined as much by the prompts it disclaims as by the prompts it claims. The anti-trigger is the executive analogue of *task-set inapplicability* — the recognition that a familiar-looking situation is, on a salient cue, the wrong situation for this program. The negative path reuses the same fuzzy/stemmed machinery as the positive path, so it inherits the same typo- and inflection-tolerance and adds no new scoring primitive.

3.4 Confidence tiers and what they gate

Raw scores are reduced to a **confidence tier** — `none`, `low`, or `high` — because the executive must behave differently depending on how sure it is, just as a person hesitates before committing to an ambiguous task framing. A tier is `high` only when the leader clears the threshold with margin *and* leads the runner-up clearly; everything else that clears the bar is `low`; and a strong anti-trigger hit on the would-be leader caps it at `low` even when it clears on positive evidence alone:

```
function CLASSIFY_CONFIDENCE(matches, threshold):
    if matches is empty:                          return "none"
    top    <- matches[0].score
    second <- matches[1].score or 0
    if matches[0].anti_vetoed:                        return "low"    # anti-trigger caps pro
    if top >= threshold + 3 and top - second >= 2:  return "high"   # clear leader
    return "low"                                                    # barely over, or a tie
```

- **High** confidence: a clear leader well above the bar, with no anti-trigger veto. Seed the plan silently and proceed.
- **Low** confidence: barely over the bar, a near-tie among candidates, or a leader that an anti-trigger flagged. Seed the best, but surface the alternatives; optionally break the tie with a short, context-shared judge call.
- **None**: nothing clears the bar. This is not a dead end — it is the *authoring opportunity* (Section 5).

3.5 Auto-seeding at turn start

The matcher runs automatically as a turn-start hook, not on the agent's initiative. This is deliberate: an executive function that must be *remembered* is an executive function that fails under load. The hook scores the prompt, builds a plan from the matched recipe(s), and persists it — with one inviolable rule: **it never clobbers an existing `in_progress` plan.** An explicit plan, or one carried over from a prior turn, always wins over a fresh auto-seed. Trivial and conversational turns (and re-injected completion signals) are filtered out before scoring, so the executive engages exactly when there is a program worth running.

The accumulating record of *which prompts produce no match* is itself a signal: a class of prompts that repeatedly fails to match means the library is too thin, the work has drifted into new territory, or it is time to author. Mining that signal is how the library grows — and Section 5 makes the growth automatic.

4. Capability 2 — Composition (Recipes Built from Recipes)

Hierarchical structure is the hallmark of executive control: complex behaviours are not flat sequences but programs that call sub-programs. The substrate gives recipes two complementary forms of composition, mirroring the two ways code reuses code.

4.1 Static composition — step-merge (`composes:`)

A recipe may declare, in its metadata, that it *composes* other recipes. At plan-build time, the composed recipes' steps are expanded **into** the merged plan ahead of the composing recipe's own steps — so a composite reads as "do sub-recipe A, then sub-recipe B, then my own steps." This is inlining: the sub-recipes' steps become first-class rows in the single resulting plan. Expansion is **cycle-guarded** (a recipe already on the expansion stack is not expanded again)

and **depth-bounded**, so a composition graph can never loop forever. Step de-duplication by normalised title keeps a step that appears in two composed recipes from running twice.

```
function BUILD_MERGED_PLAN(matches, library):
    steps <- [];  seen <- {};  expanded <- {}
    function ADD_FROM(recipe, depth):
        if recipe in expanded or depth > MAX_COMPOSE_DEPTH: return
        mark recipe expanded
        for sub_slug in recipe.composes:           # composed steps lead
            ADD_FROM(library[sub_slug], depth + 1)
        for step in parse_steps(recipe):
            key <- normalise(step.title)
            if key not in seen:
                add step to steps;  mark key seen
    for m in matches: ADD_FROM(m.recipe, 0)
    return plan(steps)
```

This is how a heavyweight workflow ("ship a paper end-to-end") is assembled from lighter, independently useful recipes ("draft," "adversarially verify," "compile") without copying their steps.

4.2 Dynamic composition — runtime sub-recipe calls (`uses:`)

Static merge fixes the program before execution; sometimes a step should *call* another recipe at runtime, the way a function call defers control to a callee. A step whose body opens with a `uses: <recipe>` directive runs that sub-recipe instead of a plain worker dispatch, on a derived child session, and folds its result back as the step's outcome. This is a true call: the sub-recipe has its own plan, its own steps, its own fan-out.

Two safeguards keep recursion safe, exactly as a runtime keeps a call stack from overflowing:

- **Depth cap.** Sub-recipe calls may nest only to a fixed maximum; exceeding it errors the step rather than recursing further.
- **Cycle guard.** The chain of recipe references on the current call stack is tracked; a recipe that would re-enter itself (directly or transitively) is refused.

A subtle but important parsing rule guards correctness: only the **consecutive leading directive lines** of a step body are interpreted as directives. A `uses:` mentioned in prose, or shown inside a code fence as an example, does *not* fire. This prevents a documentation example from accidentally triggering a recursive call — the difference between citing a function and calling it.

```
function EXECUTE_STEP(step, ctx):
    directives <- leading_directive_lines(step.body)   # stop at first prose/blank line
    sub_ref    <- parse_uses(directives)
    if sub_ref is set:
        if depth(ctx) >= MAX_USES_DEPTH or sub_ref in ctx.call_chain:
            return error("recursion guard")
        result <- RUN_RECIPE(sub_ref, child_session(ctx), chain = ctx.call_chain + [sub_ref]
        return done(result.note)
    else:
        return dispatch_worker(step)
```

Together, `composes:` (inline at build time) and `uses:` (call at run time) give recipes the full reuse vocabulary of a programming language — composition and invocation — with the loop-safety guarantees a substrate must provide.

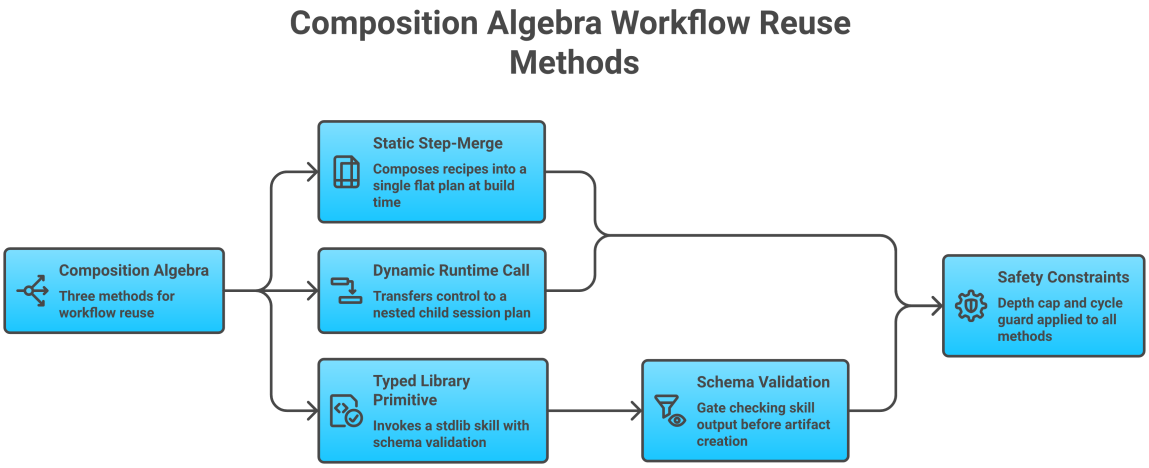


Figure 2. Composition algebra: static step-merge flattens sub-recipe steps into one plan at build time (de-duplicated, depth- and cycle-guarded); a runtime call transfers control into a nested plan on a child session and folds its terminal note back as the step's outcome; an `invoke skill:` directive calls a typed stdlib primitive whose output is schema-validated. All are bounded by a depth cap and a cycle guard.

4.3 Calling the library — `invoke skill:`

There is a third leading directive, and it completes the reuse vocabulary. Where `uses:` calls a *recipe* (another structured workflow) and `composes:` inlines one, `invoke skill:<id>` calls a typed **stdlib skill primitive** — a single vetted, reusable routine with a declared output schema, drawn from a deposited skill library rather than synthesised on the spot. The directive parses exactly like its siblings (leading-directive-lines only, so a prose or fenced mention does not fire) and is recognised by the runner's scanners regardless of directive order, so `loop:`, `uses:`, and `invoke skill:` can be combined on the same step.

The execution path injects the skill's documented procedure into the worker's task, adopts the skill's output schema (a step's own `out:` port may *narrow* it but not widen it), validates the produced artifact against that schema, and — on a validation miss — redispatches within a bounded budget before persisting the typed artifact. On a missing or deprecated skill the step **fails closed** rather than silently degrading to a free-form dispatch. The typed-output machinery this rides on (the schema, the validate→redispatch→persist path) is specified by the companion work on typed, composable recipes; here it matters only that the substrate can now invoke a vetted routine, not merely inline a sub-recipe.

The distinction is worth stating because it changes what "sequence + branch + bounded iteration" (§6.1) can express: the minimal vocabulary now includes a **library call** to a typed primitive. The executive can reach for a known-good routine the way a programmer reaches for a standard-library function, instead of re-deriving it inline every time — and because the skill is typed, its output is checkable, which a free-form sub-recipe's is not.

5. Capability 3 — On-the-Fly Authoring (Skill Acquisition)

5.1 Closing the no-match loop

The executive analogue here is skill acquisition: when a person meets a task for which they have no rehearsed program, they construct one — and they *keep* it, so the next encounter is no longer

novel. A recipe substrate that merely shrugs on no-match has no way to grow; its coverage is frozen at whatever a human last hand-authored. The substrate closes this loop two ways, and the distinction between them is real.

Authoring from intent. When matching returns `none` for a non-trivial turn, the turn-start hook injects a `recipe-gap` directive into the agent's context, instructing it to compose a new recipe specification — a slug, title, summary, tags, and three-to-seven ordered steps with tool hints and done-when criteria — and to author it. Authoring is a single call that **validates, assembles, and persists** a real recipe document. The LLM *synthesises* the steps from the prompt. From the next turn onward, that document is in the index and matchable like any hand-written recipe.

Authoring from the library. There is also a deterministic sibling: rather than synthesising steps, the substrate can *mechanically wire existing typed primitives into a runnable plan.* A compose call takes the query, searches the deposited skill library semantically, and emits one `invoke skill:` step per hit in rank order, then runs the result through the same validator and persistence path. This is authoring *from the library* — no LLM synthesis, fully determined by what the skill search returns — and the no-match flow prefers it: on a gap the substrate tries compose-from-library first (when the skill search returns relevant primitives) before falling through to from-scratch synthesis. Both stamp the result with an authored-by marker, so the §5.3 never-overwrite-curated guard holds for either path. The coverage gap heals itself, in the conversation that exposed it — sometimes by writing new steps, sometimes by wiring known-good ones.

5.2 Validation — the gate against a corrupt library

A self-authoring system that writes unvalidated files would slowly poison its own library. Authoring therefore refuses any specification that would produce a malformed or unsafe recipe:

```
function VALIDATE(spec):
    errors <- []
    if spec.slug does not match SAFE_SLUG_PATTERN:        # lowercase, no slashes, no ".."
        errors += "slug unsafe (also the on-disk dir name)"
    if spec.title or spec.summary missing:               errors += "title/summary required"
    if spec.tags is empty:                                errors += "tags required (else unr
    if spec.category set and not in CANONICAL_CATEGORIES: errors += "bad category"
    if spec.steps is empty:                               errors += ">=1 step required"
    for i, step in spec.steps:
        if step.title or step.body missing:               errors += "step {i} incomplete"
        if step.body contains a numbered markdown heading: # "### N. ..."
            errors += "step {i} body would re-parse as a PHANTOM step"
    if spec.parallelism_groups set:
        require every step index covered exactly once, in range, non-overlapping
    return errors
```

Three checks are load-bearing. The **slug** is also the directory name, so it must be traversal-safe — a self-authoring system that accepted `../../etc` as a slug would be a security hole, not a feature. The **phantom-heading** check forbids a numbered markdown heading inside a step body, because the runner splits steps on exactly that pattern; an un-checked heading would silently desynchronise the step count from the parallelism map. The **parallelism coverage** check guarantees the fan-out groups reference every step once and only once. The same validator gates *both* authoring paths and every imported recipe (§11.3), so there is a single chokepoint through which no malformed or unsafe recipe can reach the library.

5.3 Provenance and the never-overwrite rule

Authored recipes are stamped with an *authored-by* marker that distinguishes them from the curated, hand-written library. The persistence layer will overwrite an authored recipe only with an explicit overwrite flag, and **will never overwrite a curated recipe** — the substrate cannot cannibalise its own carefully written playbooks. Trigger phrases supplied by the author are folded into the recipe's tag set, because the matcher scores against tags; an authored recipe whose triggers never reached the tags would be invisible to the very matcher meant to surface it next turn. (Authored recipes are never auto-published to any shared registry; publishing is an explicit, separate human action — see Section 11.4.)

Two persistence properties keep this discipline durable. First, provenance lives in the recipe's own frontmatter — authored-by, and, for a composed recipe, the lineage of which skills and which source query produced it — so a recipe's history is a property of the artifact, not a sidecar that can drift from it. Second, a continuously-edited library cannot live only inside the packaged tree, because an upstream update would clobber operator edits; the substrate therefore keeps a writable **out-of-repo overlay** that the runner, the read RPC, and the save endpoint all honour with **read-precedence** over the bundled copy (first-readable-wins, and an in-place edit is write-redirected to the overlay). Separating the user-customised overlay (writable, precedence) from the packaged baseline (read-only, replaceable) is what lets the library evolve continuously without losing local adaptation on every sync — the substrate's "captured behaviour" survives the fork's own updates.

This is the compounding thesis applied to workflows: each authored recipe is a captured behaviour that makes the next encounter of that task class cheaper, and the library is a *growing* asset rather than a fixed one.

6. Capability 4 — Recipe Loops (The Structural Gap, Closed)

6.1 Why loops are the dividing line

A straight-line recipe is a checklist. A checklist cannot express "keep collecting findings until there are no new ones," or "retry this verification up to five times," or "iterate the draft until it crosses a quality marker." Those are the shape of real iterative work, and their absence is precisely the structural gap that separates a static playbook from a hand-coded agent workflow. A loop construct — even a bounded one — closes the gap, because sequence + branch + bounded iteration is the minimal vocabulary of a general execution substrate. With loops, recipes stop being checklists and become *programs* for behaviour.

6.2 Three bounded loop modes

A step whose leading directive is `loop:` repeats — whether the step is a plain worker dispatch, an `invoke skill:` call, or a `uses:` sub-recipe call — under one of three termination conditions:

- `loop: count N` — run exactly N times (or until an iteration fails).
- `loop: until-dry [max M]` — re-run until the worker's done-note signals it found nothing new this pass, capped at M.
- `loop: until MARKER [max M]` — re-run until a step note contains a named marker, capped at M.

The "dry" test is a small natural-language classifier over the worker's done-note: an empty note, or one containing phrases like *no new*, *nothing left*, *complete*, *exhausted*, *all covered*, counts as dry and terminates the loop. This lets a worker signal exhaustion in prose rather than via a rigid protocol.

```
function RUN_LOOPED_STEP(step, loop, ctx):
    for i in 1 .. loop.max:
        result <- EXECUTE_ONCE(step, ctx, iteration = i)
        if not result.ok:                                          break   # failure stops the
        if loop.mode == "until-dry"   and IS_DRY(result.note):  break
        if loop.mode == "until-marker" and result.note contains loop.marker: break
        # count mode: continue to loop.max
    mark step done with last result.note

function IS_DRY(note):
    if note is empty: return true
    return note matches /no new | nothing left | complete | exhausted | finished | all (cove
```

6.3 Bounded by construction — the anti-runaway guarantee

A self-driving loop in an autonomous agent is a liability unless it is *provably* bounded. Every loop is therefore clamped twice: a per-loop working maximum (a default of five iterations, or a ceiling derived from the live situation — prior iterations, the recipe's empirical success rate, and a convergence heuristic — rather than a frozen quantity), and an absolute hard ceiling of twenty-five that no recipe — authored or hand-written — can exceed. The derived bound is the principled stopping point; the frozen twenty-five is only a structural safety net (the policy of deriving bounds from the situation rather than freezing a threshold is the subject of separate work, and we adopt it here without re-deriving it). The count is clamped into `[1, HARD_CAP]` at parse time, so even a maliciously or accidentally enormous `count` collapses to the ceiling rather than running away. The substrate is expressive enough to iterate and *constitutionally incapable* of spinning forever — the executive analogue of perseveration control, the prefrontal function that stops a behaviour that has stopped being productive.

`loop:`, `uses:`, and `invoke skill:` compose on the same leading directive lines, so the most powerful single construct the substrate offers is "loop a whole sub-recipe until it reports dry" — iterate an entire nested workflow to fixpoint, bounded.

7. Capability 5 — Dynamic Effort Routing (Effort Regulation)

7.1 From classify-and-log to classify-and-drive

The prefrontal cortex meters effort: it does not spend the same metabolic budget on "pass the salt" as on "prove this theorem." An agent that reasons at full intensity on a greeting and at minimum intensity on an architecture decision is mis-regulated in both directions — wasteful on the trivial, negligent on the hard. Earlier effort routing only *classified* the turn and logged the result, which changed nothing. The substrate makes the classification **drive** the turn.

7.2 The four tiers and the signals behind them

Each turn is scored into one of four tiers — **trivial** → **standard** → **deep** → **ultra** — from cheap, language-level signals: prompt length, the number of clauses/sub-tasks (conjunctions, list markers, line breaks as a proxy for "how many things am I being asked to do"), the presence of code or file paths, the count of distinct "hard work" verbs (*architect, design, investigate, refactor, migrate, audit, debug, optimise, . . .*), the count of "maximum effort / breadth" cues (*comprehensive, exhaustive, end-to-end, entire codebase, deep dive, be generous, . . .*, weighted double), and a word-boundary-checked set of conversational/acknowledgement tokens that mark a turn as trivial.

```
function CLASSIFY_EFFORT(prompt):
    score <- 0
    if words > 40:  score += 1; if words > 120: score += 1; if words > 300: score += 1
    if clauses >= 4: score += 1; if clauses >= 10: score += 1
    if has_code:    score += 1
    if has_paths:   score += 1
    score += min(4, count_distinct(deep_verbs))
    score += min(4, 2 * count(ultra_cues))          # breadth cues weigh double
    if questions >= 3: score += 1

    if trivial_token_hit or (words <= 4 and score == 0): tier <- "trivial"
    elif score >= 7 or ultra_cues >= 2:                  tier <- "ultra"
    elif score >= 4:                                     tier <- "deep"
    else:                                                tier <- "standard"
    if tier == "standard" and count(deep_verbs) >= 3:    tier <- "deep"   # floor
    return tier
```

A trivial turn must have **no** hard or breadth verbs — the design refuses to down-classify a short prompt that nonetheless asks for hard work.

7.3 What the tier controls

The tier is not advisory; it sets four levers for the turn, injected as guidance the worker adapts to:

Tier	Model tier	Thinking budget	Orchestration mode	Token generosity
trivial	minimal	none — answer directly	solo, inline	terse (a sentence or two)
standard	standard	brief; verify before claiming done	solo, inline	normal — cover the ask, no padding
deep	maximum	extended; lay out KNOW / ASSUME / DON'T-KNOW before acting	parallel — fan independent read-only work to subagents	generous
ultra	maximum	maximum; design before implementing	full workflow — understand → design → implement → review, with adversarial verification and a completeness critic at the end	very generous — correctness over cost

For trivial turns the substrate injects *nothing* and keeps the turn cheap and fast; the executive stays dormant when there is nothing to regulate. For deep and ultra turns it shifts the whole posture of the agent — bigger model, more thinking, parallel or full-workflow orchestration, and explicit permission to spend tokens on correctness. This is the lever that lets a single agent be both a quick conversationalist and a rigorous engineer, choosing per turn rather than per deployment.

7.4 From guidance to a binding budget

The table above describes *guidance*, and guidance has no teeth: a runaway worker can ignore the token-generosity lever and keep spending. The substrate closes that gap with a per-spawn **enforced** budget. The budget itself is derived from the live situation — fan-out width set by the required-field surface, by whether the step invokes a skill, by the recipe's empirical fitness (wider when the recipe is shaky), and by token affordability — not frozen, with a structural ceiling only as a backstop. A runner watchdog recomputes the budget on each tool result and at each assistant-message start, and on exhaustion it does not guillotine the work: it **settles a partial** result, records terminal lifecycle metadata, and stops the spawn gracefully (with a `budget-exhausted` stop reason) rather than killing a near-done task. The token-generosity lever in §7.3 is therefore an enforced lever, not advice — the executive actually withdrawing metabolic budget rather than merely recommending it, with adaptive pressure before a graceful cliff. The honest seam: the tool-whitelist and budget validation land at the spawn boundary, the enforcement at the run watchdog; §11 reports this with its LIVE status.

7.5 Where a recipe may run — load constraints

A second discipline borrowed from the same agent-skills plugin (Osmani, 2026) belongs here: each of its skills declares *where* it may load — the main orchestrator versus a subagent — with a documented degraded fallback for the unmet case. The substrate dispatches the same recipe steps to the main worker and to fan-out subagents (the parallel and full-workflow modes above) with no notion that a given recipe or step might be invalid in a subagent context — a step that needs the orchestrator's privileges or its conversation-level view will simply misbehave when fanned out. A per-recipe (or per-step) **load-constraint** declaration, with a degraded fallback when the constraint is unmet, is the executive analogue of task-set *applicability*: the recognition that a program is valid in one execution context and not another. It belongs in the recipe schema (Appendix B) and is consulted on the dispatch path, so that fanning a step out is gated on that step actually being safe to run there — and falls back gracefully, rather than silently, when it is not.

8. Capability 6 — The RECIPES Observability Panel (Metacognitive Monitoring)

8.1 Why observability is a first-class capability, not a nicety

An executive that runs invisibly is impossible to trust or to debug. If a human collaborator cannot see *which* program is running, *where* it is, and *how* each worker is doing, then every long-running turn is an opaque wait, and every failure is a mystery. The substrate therefore treats observability as a designed capability with its own surface: the **RECIPES** panel.

8.2 The split-of-concerns invariant

One rule governs the panel and is the most load-bearing invariant of the whole system:

> **Chat carries substance; the panel carries orchestration mechanics. Never the reverse.**

The substantive answer — what was found, what changed, what is stuck — goes in the chat. The orchestration mechanics — which recipes were searched, which step is active, which subagent is doing what — go in the panel. Pushing dispatch chatter into the chat drowns the

substance; pushing substance into the panel hides the answer. If a person has to flip between surfaces to understand where the agent is, the split was drawn wrong.

8.3 What the panel renders

The panel renders the full life-cycle of a recipe run as several stacked sections:

- **Discovery.** Catalog size, the scored candidate recipes with their scores (including any anti-trigger penalty applied), the matching threshold, and what was selected (or a visible NO-MATCH). This makes task-set selection auditable: a wrong recipe choice — or a recipe that *should* have been vetoed by its anti-trigger — is now a thing you can *see*, not infer.
- **Composition.** When more than one recipe contributed (via match-merge or `composes:`), a summary of which recipes contributed how many steps, and how many steps survived de-duplication. Each step row in the plan carries a badge naming its source recipe, so provenance is not lost in the merge.
- **Plan with per-step status.** The ordered steps, the single active step, and each step's `pending / in_progress / done / error` status — the externalised working-memory cursor. A read-only resolver can return the *actual parsed program* (each step's title, prose, the skill it invokes, the sub-recipe it uses, its ports), so the panel renders the program the substrate is executing, not merely a trail of what it did; each step is attributed to the turn and to the skill executing it.
- **Per-step fan-out — subagent vitals.** Under the active step, one row per dispatched subagent: a stable colour derived from its run identity, its model, its status, elapsed time, and the tool it is currently using. Because fan-out is settled-not-silent (§2.1), a failed branch renders *which* unit failed and *why* (a typed error kind) rather than vanishing. This is where "the agent is working" becomes "this subagent, on this model, is reading this file, and has been for forty seconds" — and "that subagent failed, here, for this reason."
- **The provenance / decision trail.** A running log of the executive's decisions, each tagged with a small fixed vocabulary of verbs — **searched**, **matched**, **merged**, **composed**, **authored**, plus the in-flight verbs **dispatch / complete / note / transition / warn**. Reading the trail top to bottom reconstructs exactly how the substrate reasoned its way from prompt to plan to execution.

8.4 The compact-expand decision trail

A trail that grows without bound becomes noise, and a panel that stays verbose for hours after work ends is worse than no panel. The decision trail is therefore **compact by default and expandable on demand**, and it self-folds: when the last subagent completes and the call tree returns to idle, the recipe header and the accumulated trail fold away while the persistent plan (a task tracker, not an in-flight indicator) remains. The state is kept in memory for the next job's resumption; the panel simply stops painting it. The executive's monitor is loud while there is something to monitor and quiet when there is not — the metacognitive analogue of attention that disengages when the task is done.

9. Integration with the Surrounding Cognitive Systems

The substrate does not stand alone; it sits among the other organs of the agent's cognition and coordinates them. Each retains its autonomous function and gains a coordination point.

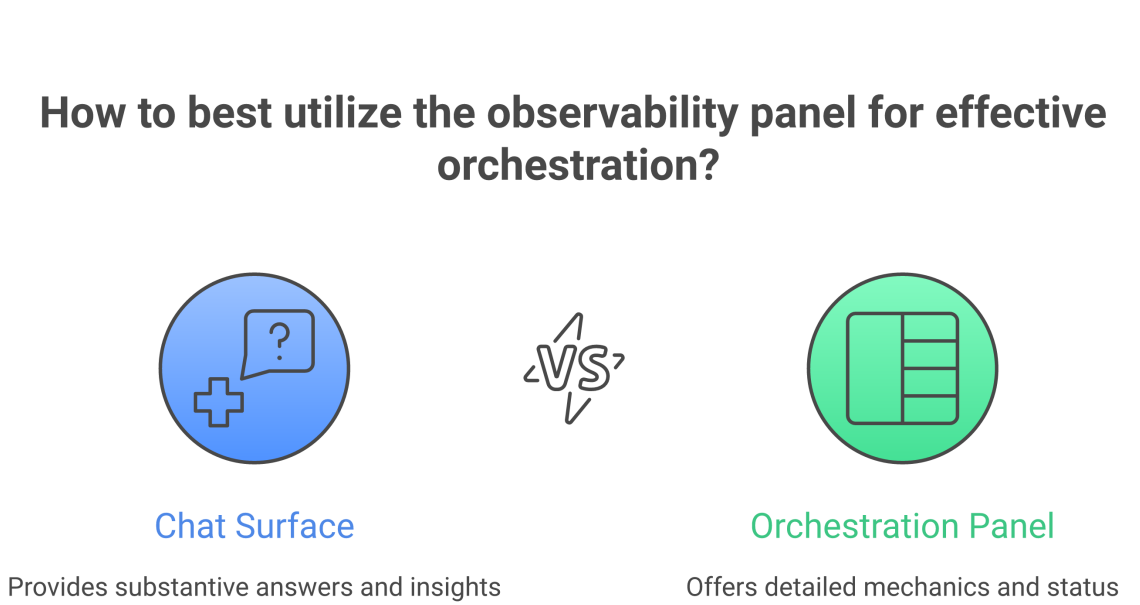


Figure 3. The RECIPES panel sits beside the chat, divided by the split-of-concerns boundary: the chat carries the substantive answer, the panel carries orchestration mechanics. The panel stacks Discovery (catalog size, scored candidates with anti-trigger penalties, threshold, NO-MATCH), Composition (source badges, de-dup count), the Plan with per-step status and the parsed program, per-step Fan-out (one subagent-vitals row per dispatch, typed failures attributed), and the compact, expandable decision trail.

9.1 Identity

A single persistent identity — personality, values, voice — is loaded into every call the substrate makes: the worker's execution calls, any tiebreak or reflection call, and every subagent dispatch. There is no separate "planner persona" to introduce voice discontinuities. The executive and the worker are the same self, wearing different instructions. This is what keeps a parallel fan-out of five subagents reading as one coherent agent rather than five strangers.

9.2 Memory

Memory retrieval supplies the context the matcher and the worker share, and the substrate uses it two ways. Recipe-contextual retrieval injects prior executions of the same recipe class into the plan ("last time this recipe ran on this module, the root cause was in the injection layer"). Reflection-driven writes push freshly learned lessons back through the indexing pipeline so they are retrievable next session. Bi-temporal validity on memory edges — facts that become *superseded* rather than overwritten — is the natural upgrade path here, so a recipe's contextual hints reflect what is true *now* rather than what was once true.

9.3 Learned action gating

A learned action-gating system (the agent's affective intuition for which operations are dangerous) sits beneath the worker's tool calls and complements the recipe's structural gates. The integration is bidirectional: gating receives recipe context — which step, which recipe, which preconditions were met — so it can learn that "a file write during the *fix* step, after a successful *diagnose* step, is safe" while "a file write with no prior investigation is suspect." Structural gates (the recipe's *done-when* criteria) and learned gates (the affective network) reinforce each other; the recipe gives the intuition a vocabulary.

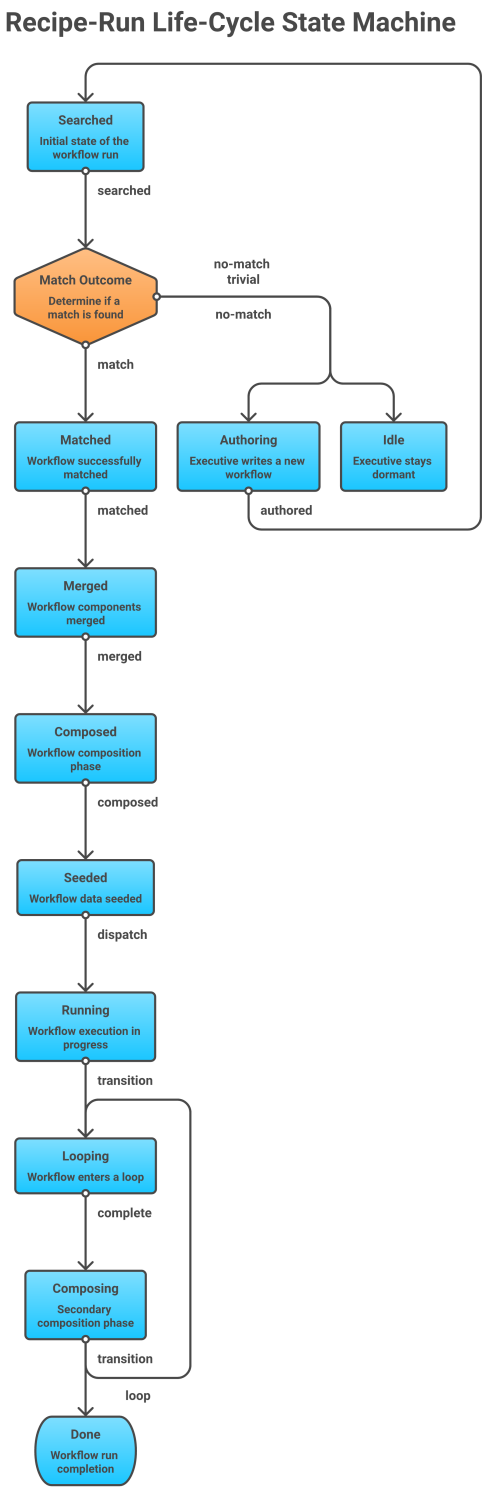


Figure 4. The life-cycle state machine the RECIPES panel renders: searched → matched → merged/-composed → seeded → running → looping/composing → done, with the no-match → authoring → matchable-next-turn branch and the no-match → trivial → idle branch. Every transition emits a provenance-trail verb, so the figure doubles as a legend for the RECIPES trail.

This integration is no longer design-only at the enforcement layer. The affective gate now *bites*: native enforcement runs as a pre-tool-use hook that can deny a tool call before it executes, even under a permissive run posture, rather than merely scoring it and logging. The consequence for the substrate's reliability story is that a recipe's structural gates and the learned affective gate are now both binding — the *done-when* discipline shapes what the worker attempts, and the affective gate can refuse the attempt at the tool boundary — so "sits beneath the worker's tool calls" describes an enforced floor, not an advisory observer.

9.4 Deliberation and budget

When a step is genuinely high-stakes, it can fan out to multiple models for a structured deliberation and synthesise the result — the recipe step is the natural activation point for cross-model debate that otherwise has no obvious trigger. And the effort router's model-tier choices are exactly where budget-aware prompting belongs: under budget pressure, the same classifier that picks "maximum" can be told to step down a tier, so the executive regulates spend as well as effort (the binding form of which is §7.4).

9.5 Reflection and self-supervised completion

Post-action reflection is no longer a turn-by-turn ritual that fires regardless of whether anything happened. The substrate scales it to the work: a full reflection (with recipe context, execution trace, and plan-versus-outcome comparison) after a substantive recipe completes; a lightweight pass after a simple task; an extended root-cause reflection after a failure — with the authority to *recommend a recipe update*. Matching reflection depth to task complexity preserves the compounding benefit of learning while eliminating the latency tax of reflecting on a greeting. Reflection is the ADAPT phase of the cycle, and its most important output is sometimes not a memory note but an improved recipe. The offline counterpart — distilling per-recipe *fitness* from accumulated execution episodes — is now wired into the matcher; Section 11.2 reports it.

The reflection phase's forward-looking sibling is *completion supervision*: judging not whether a step finished but whether the **whole task** is actually done, and re-engaging effort until it is. The substrate runs this as a critic at the end of an ultra-tier turn (§7.3), and the discipline it should follow is the subject of §9.6 — because how a completion critic is structured determines whether it can be fooled.

9.6 Doubt-driven verification — a completion critic that cannot inherit the worker's framing

A completion critic that judges the task *inside the same run and chat window* inherits the worker's framing: it sees the producing reasoning, and reasoning that talked itself into "mostly done" tends to talk the critic into it too. A loop-until-a-completion-marker critic — the natural shape, built directly from the substrate's own `loop: until MARKER` vocabulary (§6.2) — is a real improvement over no critic, but it shares this blind spot: it presses "is the whole task done?" against a context that already believes the answer is yes.

The sharper discipline, drawn from the doubt-driven-development recipe in the same agent-skills plugin (Osmani, 2026), is to put a **fresh-context** boundary between the work and its review. Its loop is CLAIM → **EXTRACT** (reduce the work to its artifact and its contract, with the producing reasoning stripped out) → **DOUBT** (a fresh-context adversarial reviewer that never saw the reasoning re-examines the artifact cold) → **RECONCILE** (each finding is classified against the artifact text, trivial versus non-trivial) → **STOP** (explicit termination when only trivial findings remain, at a three-cycle cap, or on user override, with stated criteria for what counts as a non-trivial decision). The key mechanism a same-context critic lacks is exactly that EXTRACT/DOUBT boundary: stripping the reasoning and reviewing the artifact cold is

structurally harder to fool than asking the producing context to grade itself. Its bounded STOP rule (trivial-only / three-cycle / override) is the same derived-ceiling discipline the substrate already uses for loops (§6.3), so it imports cleanly.

This fits the substrate without new machinery: doubt-driven verification is naturally a `uses:`-callable sub-recipe that composes after a worker step, so an ultra-tier plan can end with `uses: doubt-driven-verify` exactly where it would otherwise end with a same-context completion check. And it pairs with the now-enforced affective gate of §9.3: where the structural critic re-reviews the *artifact* cold, the affective gate refuses dangerous *actions* at the tool boundary — a structural completion gate and an enforced safety gate biting from two different directions on the same execution.

10. Evaluation Design

10.1 First-attempt accuracy

The substrate should raise first-attempt accuracy by structurally preventing the most common failure modes — the same way the prefrontal cortex prevents impulsive errors by holding the plan. Each gate targets a known failure:

Failure mode (executive deficit)	Mechanism that prevents it
Fixing before reproducing (impulse)	debug recipe's *reproduce* gate before *fix*
Coding before designing (impulse)	feature recipe's *design* gate before *implement*
Searching before scoping (impulse)	investigate recipe's *scope* gate before *search*
Selecting a look-alike but wrong program (mis-recognition)	a recipe's `not-when:` anti-trigger vetoing a lexical false-positive
Declaring done before verifying (premature closure)	every recipe's terminal *verify* step, hardened by a fresh-context doubt-driven critic
Stopping an iterative task early (under-persistence)	`until-dry` loops that run to exhaustion
Over- or under-investing effort (mis-regulation)	effort router driving model/thinking/tokens per tier, enforced by the spawn budget

The headline metric is the change in weighted first-attempt accuracy across a representative task mix, attributing each delta to the gate responsible.

10.2 Matching quality

Matching is evaluated on a labelled set of prompts with a known correct recipe (including paraphrases, inflections, and typos), reporting top-1 and top-3 accuracy and, separately, the **NO-MATCH precision** — how often a true coverage gap is correctly flagged versus a real match being missed. The anti-trigger mechanism is evaluated on its own axis: a held-out set of *near-miss* prompts that lexically resemble a recipe but lie outside its scope, measuring how often the `not-when:` penalty correctly demotes or vetoes the look-alike without suppressing genuine matches (anti-trigger precision/recall). The confidence tiers are evaluated by calibration: high-confidence selections should be right far more often than low-confidence ones.

10.3 Authoring quality and library health

For on-the-fly authoring: the rate at which an authored recipe is re-matched and successfully reused on subsequent turns (the loop-closure rate), the validation rejection rate (how often a draft is malformed), and a periodic audit of library health — duplicate slugs, dead recipes that never match, and recipes whose steps never satisfy their gates. Authoring-from-library (compose) is evaluated against authoring-from-intent on the same gaps: how often the deterministic, skill-wired plan succeeds where the synthesised one would, and vice versa.

10.4 Loop termination

Every loop mode is checked for the two failure directions: premature termination (an `until-dry` loop that stops while work remains) and non-termination (caught structurally by the hard cap, but worth measuring as *how often the cap is the thing that stops a loop*, which signals a mis-specified condition or a too-tight derived ceiling rather than a real fixpoint).

10.5 Cost, latency, and the artifact-compression frontier

Because matching is local and effort-gated, the per-turn overhead should be near-zero for trivial turns and small for the rest. The evaluation reports overhead as a function of effort tier and the latency the executive adds *before* visible work — which, for complex turns, is expected to be repaid many times over by fewer wrong turns and the user's ability to watch progress rather than wait blindly.

A second axis becomes measurable once the carry-forward artifact (§11.1) is compressed rather than truncated: the **compression-versus-accuracy frontier**. The relevant external reference point is the headroom context-compression project (Chopra, 2026), whose reproducible eval suite reports 60–95% token savings at roughly zero accuracy delta on standard benchmarks (GSM8K, TruthfulQA, SQuAD, BFCL) for its reversible Compress-Cache-Retrieve scheme. The substrate's artifact spine is the natural place to import that measurement: for each artifact content-type (a code done-note, a JSON tool result, a diff), report the tokens saved by the carry-forward compressor against the rate at which a later step had to *retrieve* the verbatim original — a concrete compression-vs-recovery curve, rather than the bare assertion that overhead is "small."

11. What Has Shipped Beyond the Core, and What Remains Open

Sections 3–8 specify the six core capabilities at the heart of the substrate. This section reports five harder extensions of the thesis. Four are implemented and running; the fifth — searched execution — remains specified but unbuilt. The honesty matters: a substrate that overstates its roadmap is as misleading as one that understates what it does, so we mark each with its status and the seam it attaches to.

11.1 Durable checkpointing — implemented, and now reversible

The paper's claim is that "the plan *is* the agent's working memory, externalised" (§2.3). For that to be true, the durable spine must be read back, not merely written, and it is. The runner reads an `in_progress` plan whose recipe reference matches the current run and resumes from its `currentStep` cursor, skipping any step already marked `done`; on each step's success it persists a digest of the done-note into the reserved per-step `artifact` field, and it carries the prior steps' artifacts forward as upstream context for the step that follows. A sub-recipe call bubbles its

nested plan's terminal result up as the parent step's artifact. No schema change was needed — both the `artifact` field and the `currentStep` cursor already existed.

Two policy choices keep resume safe. Resume is **opt-in, not automatic**: a bare run always force-restarts at step 0, and the runner re-attaches to an existing plan only when an explicit `resume` flag is passed *and* the recipe reference matches — so a stale plan from an unrelated prior session is never silently picked up. And a long-polling step emits a checkpoint heartbeat after a fixed interval, so a guardian can distinguish a step that is genuinely working from one that has stalled, against the per-step ten-minute wall.

The honest residual risk in earlier versions of this design was **lossiness**: a blind char-bounded truncation of the done-note can drop context a later step needs, and the only mitigation was keeping the full note in the plan body while the bounded field carried a one-way truncation. The principled fix is to make the carry-forward *reversible* and *content-type-aware*, and the external system that solves exactly this problem is headroom's **Compress-Cache-Retrieve** (CCR) scheme (Chopra, 2026): the original artifact is cached verbatim, a compressed surrogate is what flows forward, and a later step that needs the dropped detail *retrieves the original on demand* rather than living with a lossy truncation. CCR also ships per-content-type compressors — AST-aware code compression (via tree-sitter), statistical JSON-array crushing, and log/diff/text handlers — which map directly onto the substrate's artifact payloads, where a blind char-truncation is worst *exactly* when the artifact is structured (a truncated code block or JSON array loses its tail wholesale). The §11.1 digest accordingly becomes content-type-aware (compress a code artifact by its AST, a JSON artifact by array-crushing, prose by summary) with the verbatim original cached behind a retrieve seam.

The division of labour is the honest part. The substrate owns the *structural* unit — the plan/step/artifact spine, and the decision of *when and what* to checkpoint, bounded and attributed per-step against a typed plan. CCR (or any reversible compressor) owns the *content-layer* transform — compressing any artifact generically and serving its original back on demand, with a measured accuracy-delta to back the trade. What the substrate does that headroom does not: bound, attribute, and resume the artifact per-step against a typed plan. What headroom does that the substrate did not: make the carry-forward reversible and content-type-aware with a measured accuracy-delta eval. Folding CCR into the artifact seam is what turns "the plan is the program counter" from a lossy description into a lossless guarantee — and it is also what makes §10.5's compression-vs-recovery curve measurable.

11.2 The self-improving library — implemented, live end-to-end

Section 5 makes the library self-*expanding* (it authors when nothing matches) and Section 9.5 makes it self-*updating* (a reflection pass can recommend a change). The third axis is self-*improvement*: feeding real execution outcomes back into the matcher so its scoring weights are not purely hand-tuned. That loop is now closed and live producer-to-matcher, with a deliberate division of labour between subsystems. A background consolidation layer — the offline process that runs between turns to compact and learn from accumulated experience — owns the *offline* measurement: it distils a per-recipe **fitness** from accumulated execution episodes — success rate, latency, token cost. Each episode attributes to a recipe only via an explicit `recipe:<owner/slug>` attribution tag the runner stamps onto the run's event store at run start, so an untagged episode is never counted against any recipe (no false attribution); the producer seam that earlier only reached the observability trail now reaches the episode store, so the consolidation pass actually credits the outcome and the matcher reads a non-empty fitness, keyed by the full `owner/slug` so two owners sharing a slug do not cross-pollute.

The matcher reads that fitness through a single injected lookup seam and folds it in *after* the lexical base score, with the lexical base preserved as a **floor** — empirical evidence can lift a proven recipe but cannot bury a lexically-relevant one whose fitness is merely unknown or low. The precedence is strict: lexical base first (including any anti-trigger penalty, §3.3), then

the empirical-fitness adjustment, then a small clamped marketplace-popularity tie-breaker last (§11.4). A recipe with no fitness record yet contributes a neutral zero, so a cold library behaves exactly as the pure lexical matcher did. Beyond matcher weights, the library can now *grow and promote reusable primitives*, not only recipes — the same skill-library-as-code flywheel an embodied agent uses to accumulate reusable behaviours over a lifetime (Wang et al., 2023): a deposited skill library accepts new typed primitives, and a candidate is promoted only when its measured success rate clears a bar set against the library's *current* success-rate distribution (a fitness bar measured against the live distribution rather than a hardcoded threshold; the no-frozen-bound policy is owned by separate work), with a never-delete archive making the deposit reversible. The honest caveat survives: a closed plan's outcome reflects the *worker's* execution, not whether the *recipe* was the right structure — so the design treats inferred outcome as weaker evidence than an explicit verdict would be. NO-MATCH mining — bucketing the accumulated no-match log by normalised intent and proposing a recipe to author (preferring compose-from-library, §5.1) once a bucket crosses a threshold — is the remaining active-authoring step that this machinery is built to support.

11.3 External acquisition — implemented

Section 5.3 establishes `recipe/1.0` as a portable document and the substrate as a general runtime rather than a closed hand-authored set. That portability is exploited from outside. Journey kits — a near-native format — are reachable through the search/install path; the substrate also bridges **Claude Code skills**, which are a different shape (a `SKILL.md` with YAML frontmatter plus loose scripts). A dedicated adapter reads the skill's frontmatter and body, infers an ordered step list, and transpiles it into a recipe spec, which is then run through the **existing** authoring validator (§5.2) — so the slug-traversal and phantom-heading guards apply to imported content without forking the validator. Bridged recipes are written into a dedicated sandbox directory that the matcher's index scans, so an imported workflow becomes matchable like any other.

The same external ecosystem that supplies importable content also supplies *discipline* the substrate has adopted natively rather than imported as data: the agent-skills plugin's anti-trigger and load-constraint conventions (Osmani, 2026; now §3.3 and §7.5) are a case of treating a fresh open-source corpus as a source of mechanism, not merely of bridgeable artifacts. When such a corpus *is* bridged, three guards make importing untrusted content safe. Bridged recipes are stamped low-trust against the curated set, marking them distinguishable. Transpilation is heuristic — a skill documents *capability*, not always an ordered procedure — so a bad transpile is caught by the same validator that gates hand-authored recipes, and a malformed skill is rejected before any file is written. And because the sandbox-path resolver blocks `../` but not symlinks, the importer refuses to follow a symlinked `SKILL.md` (or any symlinked component of its path) rather than read through it. Namespacing by owner keeps an imported `code-review` from shadowing a curated one. The one genuinely open question is policy, not engineering: whether bridged recipes are matchable on import or quarantined until a human promotes them.

11.4 The versioned marketplace — implemented (client side)

Treating a recipe as an artifact culminates in making a tested one a versioned, discoverable object rather than a file in one workshop. The frontmatter carries owner, version, tested-harness, and authored-by, and the publish path POSTs to a remote endpoint; the *semantics* on top of that plumbing live in a marketplace facade. Publishing bumps the frontmatter version under a semver policy and refuses a republish that would overwrite an already-published version — versions are immutable, and a bad recipe is yanked, not silently replaced. Install accepts a version constraint (latest / exact / a caret-tilde-or-`>=` range) resolved against the published set, with a roughly one-hour cache TTL so updates propagate without hammering the API.

The design's contract under failure is the load-bearing part: when the marketplace fetch fails, resolution degrades to the last cached copy — within or even past the TTL — and **never throws**, so an outage degrades a match to local cache rather than hard-failing a turn. Discovery folds a small **clamped** popularity bonus into the matcher score through the *same* post-base adjustment seam as the fitness feedback (§11.2), so there is exactly one place that perturbs base scores. The bonus is additive and hard-clamped to a tiny band — on the order of a fifth of a point against a scale where a single exact tag hit is worth several — with rating (0–5) supplying most of it and download count adding only a gentle log-scaled nudge. A merely-popular recipe can therefore never override a genuine text mismatch, an anti-trigger veto, or out-rank a recipe with stronger empirical fitness. Authored recipes are still never auto-published — publishing stays an explicit human action (§5.3) — and only the frontmatter owner may publish a new version of a slug. The marketplace *backend* itself is a separate deliverable outside the substrate; this is the client-side half.

11.5 Bounded search — the one open gap

The unbuilt extension is to make the plan *searched* rather than fixed — the agent analogue of deliberate, branch-and-evaluate problem solving (Tree of Thoughts; Yao et al., 2023) and of tree search that unifies reasoning, acting, and planning over a language agent's trajectories (Language Agent Tree Search; Zhou et al., 2023).

This is the most ambitious extension and the only one of the five that remains unbuilt. Today the substrate executes a *single* pre-seeded plan top to bottom; loops retry the *same* step to a fixpoint and `uses:` recurses a *fixed* sub-recipe. None of this *searches*. The unrealised degree of freedom is: when a step fails, explore alternate recipes, alternate parameters, or alternate orderings, score the branches, backtrack, and prune — turning the plan from a fixed program into a searched one, and advancing the thesis from "the substrate runs a recipe" to "the substrate finds the recipe that works."

Some of the raw material is already present: the matcher returns a ranked top-N with confidence, so "alternate recipes" is just consuming the list rather than only its first element; the per-step artifacts from §11.1 are the snapshots a backtrack would restore (and, now that they are reversible CCR pointers, restoring a branch's full pre-state is lossless rather than approximate); and the top-N typed skill primitives from the compose path (§5.1) are a richer branch frontier than recipes alone. The expensive parts are what remain unwritten — a search controller that maintains a scored frontier, a runner that can restore plan state to re-enter a prior step, and a value estimate good enough that search converges faster than brute force. The risks are why this is last. Combinatorial blow-up across recipes × parameters × orderings makes a hard budget — max nodes, max tokens, max wall-time — mandatory, not optional. It also interacts badly with loops: a long loop *inside* a searched branch can stall for the better part of an hour. Every backtrack is a real subagent dispatch, so a weak value estimate makes search net-negative. And searched execution is non-deterministic, which collides with the marketplace's need for a recipe's behaviour to be a function of its text — so a deterministic mode that disables search would be the contract for published recipes. A failure-only mode (search only when a step fails) is a reasonable, cheaper first cut, but it is not the full vision.

11.6 The boundary, stated plainly

The six core capabilities (Sections 3–8) and four of these five extensions are implemented and running. Only searched execution is specified and not yet built. Drawing that line precisely is the point: a substrate that overstates what it does is no more trustworthy than an agent that declares victory before it verifies — which is the very failure the substrate exists to prevent.

12. Conclusion

The history of engineering is the history of structure replacing improvisation: not by making practitioners smarter, but by giving their work a shape that makes good outcomes the default. The prefrontal cortex did the same thing for cognition long before engineering existed — it is the structure that lets a capable brain act coherently rather than impulsively, by holding the plan, selecting the program, sequencing the sub-routines, suppressing the wrong move, watching the result, and metering the effort.

A deployed language agent has the capability of a frontier model and, without an executive, the coherence of a reflex. This paper has argued that the missing executive is best built as a **recipe execution substrate**, and that the substrate becomes genuinely executive only when it can do six things: match intent to a program reliably and with calibrated confidence — claiming the prompts it serves and disclaiming the ones it must not; compose programs from programs, statically, dynamically, and by calling typed library primitives, with the safety rails a call stack requires; author a new program when none fits — by synthesis or by mechanically wiring the library — and keep it; loop a program under a bounded condition; regulate the effort each turn deserves and drive the turn accordingly, with a binding budget rather than mere advice; and make the whole thing observable without drowning the answer.

Of these, the loop is the quiet pivot. Sequence and selection make a checklist; sequence, selection, *and bounded iteration* make a substrate expressive enough to encode the iterative, exhaust-the-search, retry-until-clean work that real tasks demand — exactly the structural gap that separated a static playbook from a hand-coded workflow, now closed. And on-the-fly authoring is the quiet flywheel: a substrate that writes its own missing programs grows its competence inside the very conversations that expose its gaps. With durable, now-reversible resume, an empirically self-improving library running live end-to-end, external acquisition that mines fresh open-source corpora for both artifacts and discipline, and a versioned marketplace running on top of those six capabilities, the substrate realises most of the full thesis, leaving only searched execution as a specified, deliberately bounded gap.

The intern who never repeats a mistake writes things down. The intern who gets promoted writes *recipes* — programs that make the next person as effective as they are. The agent described here writes its own recipes, composes them, loops them, runs them at the right intensity, doubts its own "done," and shows its work. That is not a better worker. It is an executive.

13. The Surrounding Cognitive Stack

The substrate is the executive, not the whole mind. It assumes a small set of neighbouring systems and integrates with each at a named seam; this section states what it depends on so the architecture is reproducible without reference to any other system's internals. None of the dependencies are exotic — each is a capability a serious deployed agent already needs for its own sake.

- **Episodic memory and consolidation.** A retrieval layer supplies the context the matcher and worker share, and an offline consolidation process — the background pass that runs between turns to compact experience and learn from it — owns the recipe-fitness measurement the matcher reads (§11.2) and the skill-library deposit/promote substrate. The ADAPT phase writes its lessons back through this layer's indexing path.
- **A fast, cache-friendly index over memory.** The auto-seed hot path (§3.5) must add negligible latency, so it depends on retrieval that is structured and cache-friendly rather than a fresh similarity search each turn. Bi-temporal validity on stored facts — superseding

rather than overwriting — keeps a recipe's contextual hints reflecting what is true now (§9.2).

- **A reversible context-compression layer.** The artifact carry-forward (§11.1) depends on a compressor that caches the original verbatim and serves it back on demand — a Compress-Cache-Retrieve facility, content-type-aware — so the externalised working memory is lossless rather than truncated.
- **A learned action-gating system with native enforcement.** An affective intuition for which operations are dangerous sits beneath the worker's tool calls and complements the recipe's structural gates; it denies at the tool boundary, not merely scores, and the recipe context it receives (which step, which preconditions met) gives that intuition a vocabulary (§9.3).
- **Multi-model deliberation.** A facility to fan a decision out to several models and synthesise the result; a high-stakes recipe step is its natural trigger (§9.4).
- **Budget-aware prompting.** The effort router's model-tier choice (§7.3) is where a budget governor applies: under pressure, the same classifier that picks the maximum tier can be told to step down one; the enforced per-spawn budget (§7.4) is the binding form.
- **A typed-composable recipe language.** The typed `invoke skill:` output schema, the validate→redispatch→persist path, and the classified-error taxonomy behind settled fan-out are specified by the companion typed-recipe-language work; the substrate consumes them at named seams (§4.3, §2.1) without re-deriving them.

Between tasks, an idle agent with any intrinsic-motivation drive has an obvious use for the accumulating NO-MATCH log (§3.5): mining it to *propose* recipes worth authoring — by compose-from-library where the skills exist, by synthesis otherwise — before they are next demanded.

References

1. Osmani, A. (2026). *agent-skills: A discipline plugin for agent skills, with explicit anti-triggers and load-constraint declarations.* Open-source project (GitHub: addyosmani/agent-skills).
2. Chopra, T. (2026). *headroom: Reversible Compress-Cache-Retrieve context compression for language agents.* Open-source project, Apache-2.0 (GitHub: chopratejas/headroom).
3. Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., & Anandkumar, A. (2023). *Voyager: An Open-Ended Embodied Agent with Large Language Models.* arXiv:2305.16291.
4. Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., & Narasimhan, K. (2023). *Tree of Thoughts: Deliberate Problem Solving with Large Language Models.* arXiv:2305.10601.
5. Zhou, A., Yan, K., Shlapentokh-Rothman, M., Wang, H., & Wang, Y.-X. (2023). *Language Agent Tree Search Unifies Reasoning, Acting, and Planning in Language Models.* arXiv:2310.04406.
6. Miller, E. K., & Cohen, J. D. (2001). *An Integrative Theory of Prefrontal Cortex Function.* Annual Review of Neuroscience, 24, 167–202.

Appendix A — The Turn, End to End (Pseudocode)

```
function HANDLE_TURN(prompt, context, library):
    # PLAN: effort regulation
```

```
    tier <- CLASSIFY_EFFORT(prompt)
    if tier != "trivial": inject_effort_guidance(tier); arm_spawn_budget(tier)
    if tier == "trivial": return answer_inline(prompt, context)   # executive dormant

    # MATCH / COMPOSE / AUTHOR: task-set selection
    if context.has_in_progress_plan():
        plan <- context.resume_plan()              # never clobber an explicit/prior plan
    else:
        result <- MATCH_RECIPES(prompt, library)   # fuzzy + anti-trigger + confidence + fit
        emit_trail("searched", catalog_size)
        if result.confidence == "none":
            emit_trail("no-match")
            spec <- COMPOSE_FROM_LIBRARY(prompt) or AUTHOR_RECIPE(prompt)  # library-wire, e
            if VALIDATE(spec).ok:
                library.add(spec); emit_trail("authored", spec.slug)
                plan <- seed_plan_from(spec)
            else: plan <- implicit_two_step_plan()
        else:
            emit_trail("matched", result.matches)
            plan <- BUILD_MERGED_PLAN(result.matches, library)   # composes: inlined
            if plan.from_multiple(): emit_trail("merged", plan.sources)
            persist(plan)

    # EXECUTE (loops + sub-recipe calls + skill invocations): sequencing
    for group in plan.parallelism_groups:          # groups run sequentially
        dispatch_all(group) in parallel:           # steps within a group fan out
            for step in group:
                if not load_constraint_ok(step, here): step <- degraded_fallback(step)
                loop <- parse_loop(step); sub <- parse_uses(step); skill <- parse_invoke_ski
                if loop:       RUN_LOOPED_STEP(step, loop, context)
                elif sub:      emit_trail("composed", sub); RUN_RECIPE(sub, child(context))
                elif skill:    INVOKE_SKILL(skill, step.out, context)   # typed, schema-vali
                else:          dispatch_worker(step)
                # OBSERVE: panel updates with step status + subagent vitals (settled, attrib
                # persist a reversible (CCR) artifact pointer of each done step (resume spin
        barrier()  # survivors settle; typed failures attributed, not dropped
    response <- aggregate(plan)

    # ADAPT: learning, completion supervision, regulation
    if tier == "ultra": RUN_RECIPE("doubt-driven-verify", child(context))  # fresh-context c
    reflection <- REFLECT(scaled_to = tier, trace = plan.trace)
    persist_lessons(reflection.memory_writes)
    if reflection.recipe_update: library.update(reflection.recipe_update)
    close_plan(plan)
    return response
```

Appendix B — Recipe Document Schema (Illustrative)

```
--- frontmatter ---
schema:    recipe/1.0
```

```
slug:      debug                          # also the on-disk directory name; traversal-safe
title:     Debug & Fix
summary:   Systematic debugging - reproduce, diagnose, fix, verify
tags:      [bug, error, crash, broken, not working, fails, exception, fix]
not-when:  [design from scratch, greenfield, new feature, brainstorm]   # anti-triggers (neg
composes:  []                             # static step-merge (recipes built from recipes)
loads-in:  [main, subagent]               # load constraint; degraded fallback if unmet
authoredBy: curated                       # curated | jarvis-on-the-fly | cc-bridge
parallelism:
  groups: [[0],[1],[2],[3]]              # 0-based; serial chain (data-dependent)

--- body ---
### 1. Reproduce
   done-when: the exact failure is described with evidence
   Read the error. Run the failing case. Identify the trigger conditions.

### 2. Diagnose
   done-when: root cause identified with file:line
   Trace the cause through code; follow evidence, do not guess.

### 3. Fix
   done-when: minimal change applied
   Smallest fix that addresses the root cause. One concern per change.

### 4. Verify
   done-when: the original error is gone and tests pass
   loop: until-dry max 3                # re-run verification until nothing new fails
   uses: doubt-driven-verify            # fresh-context adversarial review of the artifact
   Run the failing test, then the broader suite, then trigger the original scenario.
```

(In a composite recipe, a step body might instead open with `uses: adversarial-verify` *to call that recipe as a sub-routine, with* `invoke skill: extract-json-field` *to call a typed library primitive, or with both* `loop: until-dry` *and* `uses: <recipe>` *to iterate a whole nested workflow to exhaustion.)*

References