SALIENCE: The Death of Fixed Thresholds, the Pyramid of Significance, and Cheap Traversal as the Basis of Next-Generation Vibe Programming

Oscar Serra, Jarvis · Independent Research

14 June 2026 — v1.2

Abstract

The salience network in the human brain does not fire on a fixed rule. It continuously re-computes, moment to moment, *what is relevant right now* — which interoceptive signal, which external stimulus, which goal deserves the next unit of attention — and switches the rest of cognition between its default and executive modes accordingly. It is the organ of *live relevance*. We take that as the design thesis of this paper: **relevance must be re-computed from the live situation, not read off a frozen list.**

This paper makes three connected arguments about how a capability-first agent harness should be built. First, **the death of fixed thresholds and fixed lists.** Hardcoded numbers, frozen rankings, and round-number caps are legacy artifacts of the primitive programming era; each one silently encodes a past world-state and goes stale as the world moves, quietly making the system obsolete. The successor is *salience* — derive every decision from the live situation at the moment of use, and re-evaluate continuously. Second, **the pyramid of significance.** Knowledge and code are organized as a fractal pyramid ordered by significance: base principles at the apex, then structural optics, then derived rules, then diagrams and flows, then code — each level deriving from the one above and forbidden to contradict it, significance decreasing and detail increasing as you descend. Third, **cheap code-traversal as the basis of next-generation vibe programming.** A pyramid is only worth building if an agent can traverse it cheaply; the plugins and structures that make navigating code and knowledge cheap — a pre-compressed documentation map, a knowledge graph, deferred-tool loading, paired read surfaces, an intent router, and reversible compression of what is pulled into context — are the real foundation of good vibe programming, because they let the agent spend its tokens on *thinking* rather than on re-reading the repository.

We then describe the executable embodiment of all three: a **native orchestration runtime** (`agent/parallel/pipeline/phase` over an existing subagent and recipe-runner substrate, schema-validated, with concurrency and budget derived from the live situation rather than from fixed caps). We are explicit that this runtime is **planned, with its derived-bound substrate now partly live** — its concurrency, retry, supervision, and fan-out bounds are computed from the live situation at real call sites today, while the four orchestration primitives themselves remain specified and borrowed. The honesty is the point: a paper that overstates its system is the documentation equivalent of an agent that declares victory before it verifies.

Keywords: salience, fractal design, fixed-threshold anti-pattern, significance pyramid, knowledge graph, cheap traversal, reversible compression, vibe programming, orchestration runtime, reversibility, observability

1. Introduction — Relevance Is Computed, Not Looked Up

1.1 The salience analogy, taken seriously

The brain has an attention problem that no fixed rule can solve. At every instant a flood of signals — a sound, an ache, a half-finished intention, a face in the crowd — competes for a finite pool of processing. The salience network (chiefly the anterior insula and dorsal anterior cingulate) is the arbiter: it integrates interoceptive and external signals and computes, continuously, which of them is *salient enough right now* to seize control, and it toggles the brain between its inward default mode and its outward executive mode based on that live computation. It does not hold a

list of "important things." A list would be obsolete the moment the situation changed — a rule that said "always attend to loud noises" would have you flinch at every door while the actual threat walked in quietly. Relevance is re-derived from the moment, not retrieved from a table.

This is the exact failure mode of software built in the primitive programming era. A hardcoded threshold is a list of "important things" frozen at authoring time. It was right once, for the world the author could see. Then the world moved, and the threshold kept firing on the old world. The salience network never has this problem because it never freezes its answer. The thesis of this paper is to give an agent harness the same property: **compute relevance live; never look it up from a frozen artifact.**

This is not a new idea. A ratified design principle already states it in one sentence: "Fixed lists, fixed thresholds, and fixed limits are legacy artifacts of the programmatic era; in a fast-moving world they go stale and make us obsolete. Decisions are derived from the actual situation at the moment of use and adapt continuously." This paper expands that principle: it argues *why* it is right, gives concrete before/after instances, derives the discipline that makes it safe to follow aggressively, and connects it to the two structural ideas — the pyramid and cheap traversal — that make a fractally-reasoned system tractable rather than chaotic.

1.2 Why one principle needs three pillars

A reader could object that "derive decisions from the live situation" is a slogan, not an architecture. The objection is fair, and the three pillars are the answer to it.

If you tear out every fixed threshold and replace it with a live computation (Pillar 1), you have created a system whose behavior is now a function of *everything* — the live model leaderboard, the real remaining budget, the latest prompting standards, the current recipe library. That is more correct and far harder to reason about. Two things make it tractable. The first is **significance ordering** (Pillar 2): if the live computations are arranged in a pyramid where each derives from a small, stable apex and never contradicts it, then "everything is live" does not mean "anything goes" — the apex pins the invariants and the churning base inherits them. The second is **cheap traversal** (Pillar 3): a system whose every decision is computed live must *read* a great deal at decision time, and if reading is expensive the live-computation discipline collapses back into caching stale answers — which is a fixed list with extra steps. Cheap traversal is what makes live computation affordable enough to actually do.

So the three pillars are not three topics. They are one principle (compute relevance live) plus the two structural preconditions that make it survivable at scale: order it by significance; make traversal cheap. The orchestration runtime in Section 5 is where all three meet in running code.

1.3 What this paper is not

This paper is not a re-derivation of the recipe execution substrate, nor of the theory of intermediate abstractions; it cites both and summarizes the borrowed ideas where they are used. It is also not a license to remove every constant in the codebase. The discipline is now a ratified design law with a precise typology (Section 2.4): a **categorical capability or permission boundary** — a principal may or may not do X, an API-required field, a security gate, a tool whitelist — stays *hard*; only a **quantity threshold** — at most N, after T seconds, up to D deep — must be made *soft and derived*. The line is not "no numbers"; it is "no quantity bound that encodes a *past judgment about a moving world* as its working value."

2. Pillar 1 — The Death of Fixed Thresholds and Fixed Lists

2.1 The anti-pattern, named

A **fixed threshold** is a hardcoded number that gates a decision. A **fixed list** is a frozen enumeration that a decision indexes into — a ranking, a leaderboard, an allow-list of "the good options." A **fixed limit** is a round-number cap on a resource. They are the same anti-pattern wearing three hats:

> A fixed artifact encodes, at authoring time, a judgment about a world-state. The world-state then changes, and the artifact keeps enforcing the old judgment. The system does not break loudly; it drifts silently into obsolescence, because the artifact still *runs* — it just runs against a world that no longer exists.

This is worse than a bug, because a bug announces itself. A stale threshold produces plausible-looking output forever. The `70%` cap that was the right burn limit when a quota reset weekly and the work was light is still `70%` after the quota doubled and the workload tripled — and nobody notices, because the system never errors. It just leaves capability on the table, or spends past the real limit, on every decision, quietly.

2.2 Why this is structural, not a code-quality nit

The reflex defense of a magic number is "it's just a constant, we'll tune it later." But "later" rarely comes, and even if it did, tuning replaces one frozen judgment with a newer frozen judgment that will itself go stale. The problem is not that the number is *wrong*; it is that the number is *fixed* in a domain that *moves*. The model landscape changes monthly. Pricing and quota terms change — programmatic-usage metering and two model-ID retirements both land on a single near-term date, so a hardcoded model ID or a hardcoded rate is a time bomb with a known fuse. Prompting best-practice changes with every model generation. Against a moving domain, a fixed artifact is not a value that needs tuning; it is a *category error* — it represents a dynamic quantity with a static one.

The salience network is the existence proof that the alternative is viable: a control system *can* re-derive its gating decisions continuously rather than store them. The cost is that the decision becomes a small live computation instead of a constant lookup. Section 2.4 argues that this cost is low, and Pillar 3 makes it lower still.

2.3 Before / after, from this project

Abstract arguments about freshness are cheap. Here are concrete instances in the harness — each a real fixed artifact, each with the salience-style replacement the principle mandates. The first four are named at the level of the principle; the next subsection (§2.4) reports the ratified design law that generalizes them and the first family of derived bounds that now run at real call sites.

(a) Model choice — from a hardcoded rank to a live sense. The auth-routing layer's current source of truth for which model to prefer is a static field, `agents.defaults.models[<provider/model>]` in the harness config (the field is live there today — `rank: 2`, `rank: 3`, and so on). That is a frozen leaderboard. The day a better or cheaper model ships, the rank is wrong, and it stays wrong until a human hand-edits the config. The salience replacement is a *live sense of what is best* — a periodically self-refreshing signal that ranks available models by current capability, price, and availability, so model choice is read from the world's present state. The ratified principle states the target directly: "model choice comes from a live, self-refreshing sense of what's best — never a hardcoded ranking." The `rank` field is the legacy artifact; the live sense is the named successor, not yet shipped.

(b) Budget — from a fixed `70%` to a reasoned allowance. The failover logic currently gates the flat-rate primary on `<70%` of the seven-day quota burned (the literal `<70%` is in the

auth-routing optic today). Seventy percent is a round number encoding a guess about how much headroom is "safe," made once, against one assumption about workload and reset cadence. The salience replacement reasons against the *actual* remaining allowance, the *actual* time to the next reset, and the *value of the work in front of it*: with a comfortable surplus and a reset imminent, spending the surplus down is welcome; with a thin allowance early in the window, the same nominal percentage should hold back. The budget doctrine says exactly this: "the cap is reasoned fractally — actual remaining allowance, time to reset, value of the work — never a fixed threshold." The `70%` is the artifact; the fractal computation is the successor.

(c) Prompting — from a pinned model version to latest-model standards. A prompt written to one model generation's quirks is a fixed list of "things this model likes." The next generation moves the target and the pinned prompt underperforms silently. The salience replacement follows the *latest* model standards — system-prompt structure, tool-use conventions, thinking-budget idioms — tracking the current generation rather than the one the prompt was born against. The principle: "Prompts follow the latest model standards . . . never pinned to one model version." This one is live as posture: no single pinned-version artifact owns the prompt.

(d) Recipes — from gated change to free evolution with reversibility as the net. The most aggressive instance, and the one that most tests the principle. A conservative system would gate every change to its own behavioral library behind review. This harness does the opposite: recipes evolve continuously and autonomously, and the safety is not a gate but *reversibility* — never-delete archives, rollback, kill-switches — plus *observability*. The autonomy doctrine makes the trade explicit: reversibility is what earns aggressive autonomy, and recipes are required to "evolve continuously and autonomously — made safe by reversibility and observability, not by constraints on change." A fixed gate is itself a frozen judgment ("this class of change is always dangerous"); replacing it with reversibility lets the system be as adaptive as the situation allows while keeping every change undoable. This is the death-of-fixed-artifacts move applied to *governance*: the gate is the fixed artifact; reversibility-plus-observability is the live successor.

The pattern across all four: a constant or a frozen list is replaced by a small computation over the live situation, and the safety the constant used to provide is re-provided by reversibility and observability rather than by the constant's rigidity.

2.4 The discipline — a ratified design law, not a case-by-case test

In v1.1 this section read as a four-part discipline the author had to apply by hand. Since then the principle has been promoted to a **single-owner, executable governance artifact**: **design-principle #19, "Derive bounds from the live situation; a hard quantity threshold is a bug,"** with a one-line corollary at the apex constitution. This is Pillar 1 and Pillar 2 meeting in one act — the principle the paper expands now *lives at the apex with a single owner*, and every optic's local "don't regress on this magic number" note points back to it. The discipline below is the law's content; the law is what makes it enforceable rather than aspirational.

The categorical-vs-quantity typology. The crisp successor to "no number that encodes a past judgment about a moving world" is a typology, not a test the reader must re-derive each time. A **categorical capability or permission boundary** stays HARD: a principal may or may not do X; an API-required field; the PII split; a security gate; a tool whitelist. A **quantity threshold** must be SOFT and DERIVED: at most N, after T seconds, up to D deep, fan out to W. Timeouts, retry counts, loop caps, recursion depth, concurrency, and cache or result sizes are all quantity thresholds — the §2.3 model/budget/prompting cases are now special cases of one law that covers the full taxonomy of bounds.

Ceiling, not working value — the cliff becomes a ladder. The law sharpens "kill the fixed cap" into a more precise truth: a frozen number is at most a *safety ceiling*, never the *working value*. The runtime value is derived from the live situation and capped *by* the ceiling; as it approaches, the system emits **adaptive pressure** — a trail or warn event — rather than

hitting a cliff, and on exhaustion it **degrades gracefully** (persist the partial, warn, continue) rather than hard-aborting in-flight work. Cliff-to-ladder is the operative refinement: the bound still exists, but it stops being a guillotine.

The derive → validate → enforce → catch chain. "Derive the bound" is only one of four cooperating layers, and the safety the old fixed cap used to provide is re-provided by the other three — the §2.3 thesis ("safety re-provided by reversibility and observability, not by the constant's rigidity") instantiated as a concrete, reusable pattern:

1. **Derive** the working bound from the live situation — a pure function whose output responds to live inputs (fitness, remaining budget, gap-to-done), never collapsible to a literal without re-introducing the frozen MAX the paper condemns.
2. **Validate** at the trust boundary — reject a malformed budget (non-integer, negative) as an invalid request, and normalize the tool whitelist; a *categorical* gate guarding the *quantity* path, the two #19 kinds side by side.
3. **Enforce** at runtime — an attempt-loop watchdog trips when a spawn meets either its token or its tool-call cap and stamps a `budget-exhausted` reason; the whitelist is enforced by filtering the tool set, so the derived budget is real, not advisory.
4. **Catch** the limit gracefully — fan-out returns catchable partials, a budget hit surfaces as a classified non-recoverable error, and null-isolation drops the exhausted leg rather than poisoning the next stage — "degrade, don't hard-abort" realized in the fan-out type itself.

This chain answers the conservative engineer's objection — "a soft bound is no bound" — by showing the bound is *enforced* (layer 3) and *survivable* (layer 4) without being *frozen* (layer 1).

How to keep it cheap. A live computation on every decision is only viable if it is cheap. Three techniques: (i) **cache with a freshness bound** — a self-refreshing model sense recomputes on an interval, not on every call, so the hot path reads a recent cached value; the freshness bound is the antidote to "caching = stale list," because the cache *expires*. (ii) **compute locally** — budget headroom and effort tier are token arithmetic over data already in hand, no model call. (iii) **reuse context** — when a live decision genuinely needs a model call, share the turn's context so that, under context-caching pricing, the marginal call costs a fraction of a fresh one.

The trust model behind the aggression. The reason a conservative engineer reaches for a fixed cap is fear that a live, self-modifying decision could go wrong. The law's closing clause states the answer as a stance, not a hope: **budgets are soft signals, not hard cliffs; there are no stoppers; reflex safety-hooks are rejected.** The justification is the trust model — code that runs as a *single trusted principal in its own gateway* gains no safety from a quantity gate, only friction. Where §2.3(d) argues reversibility-not-a-gate for recipe evolution, the law generalizes it to *all* quantity bounds: the working value is derived, the agent is trusted to run up to the ceiling, and adaptive pressure (never a stopper) is what it feels as it approaches. Reversibility — never-delete archives, kill-switches, rollback — converts "this might be dangerous" from a reason to *forbid* into a reason to *watch*, and observability is what lets a human or the agent's own reflection layer catch a live computation drifting *before* reversibility has to clean up after it. Reversibility is the net; observability is seeing the fall coming.

2.5 The anti-pattern is a population-level property — a whole-system audit

The claim that hardcoded thresholds drift a system silently into obsolescence (§2.1) is no longer asserted from four hand-picked cases. A **systematic audit of the live harness against design-principle #19** catalogued **17 prioritized real instances** of the exact anti-pattern — each with file and location, the principle violated, a severity, and the derived-replacement formula. The roll-call is instructive: a `120s` idle kill that terminated a legitimate tool-heavy turn (the bible itself called it "mask the symptom"); a `HARD_LOOP_MAX=25` that can amputate a near-converged recipe; a `MAX_USES_DEPTH=3`; a five-minute guardian that kills in-flight work; frozen

`maxConcurrent/maxChildrenPerAgent`; retry counts that ignore the provider's `Retry-After`; a whisper `timeoutSeconds=90` that would guillotine a thirty-minute transcription. This is "it drifts silently" turned into a *measured backlog with a uniform fix shape* (ceiling-not-working-value + derivation formula + adaptive pressure).

Equally important for the paper's honesty discipline, the audit also lists the constants it **deliberately left hard** — the PII grep gate, the browser direct-CDP guard, architect-only paths, API-required params. That is the categorical-boundary side of #19's typology, *demonstrated* rather than merely asserted: the law does not delete every constant; it sorts each into "quantity → derive" or "category → keep," and proves it did so on real code. Seventeen cliffs, one fix-shape, with the categorical exceptions named: Pillar 1 is a population-level property of the codebase, not a rhetorical device.

3. Pillar 2 — The Pyramid of Significance

3.1 The shape

Knowledge and code are organized as a **fractal pyramid ordered by significance.** The five levels, apex to base:

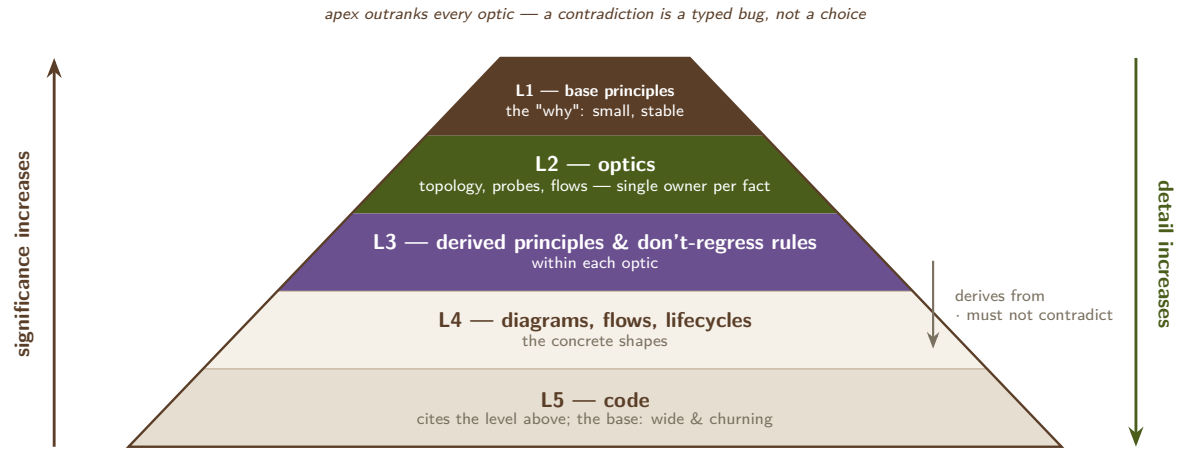

Figure 1. The pyramid of significance: five levels from a small stable apex of base principles down to a wide churning base of code, with significance decreasing and detail increasing as you descend; each level derives from the one above and may not contradict it.

- **L1 — base principles** (the "why"): the apex, small and stable.
- **L2 — optics** (topology, probes, flows, . . .): structural facts, single owner per fact.
- **L3 — derived principles and don't-regress rules** within each optic.
- **L4 — diagrams, flows, lifecycles**: the concrete shapes.
- **L5 — code**: cites the level above; the level above cites it (the base: wide and churning).

Two gradients run down the pyramid in opposite directions. **Significance decreases** as you descend: the apex holds enduring intent ("be as useful as possible; think fractally"), the base holds the most ephemeral detail (a specific function's implementation). **Detail increases** as you descend: the apex is a few hundred lines, the base is the entire codebase. The apex is small and stable precisely because it is significant — a principle that changed weekly would not be a principle. The base is wide and churning precisely because it is detailed — code changes every commit. This is the same shape at every zoom level: a single optic is itself a small stable header over wide churning facts. That self-similarity is why it is *fractal* rather than merely *layered*.

3.2 The governance — derivation and non-contradiction

The pyramid is not just a filing scheme; it is a *governance* structure with one hard rule: **each level derives from the one above and must not contradict it.** The apex states this as authority: it "OUTRANKS every bible optic. When an optic's detail contradicts this file, THIS file wins and the contradiction is flagged for repair — never silently followed." The index that routes to it makes the metaphor explicit: the index is the map; the apex is the constitution.

Three consequences follow, and they are what make the pyramid more than decoration:

1. **A contradiction is a bug, not a choice.** When an optic's detail disagrees with the apex, you do not pick which to follow. The apex wins, the optic is wrong, and the discrepancy is filed for repair. This converts "documentation drift" from a vague malaise into a *typed defect* with a defined resolution.
2. **The apex is read first and trusted most.** The index's step zero is "read the apex first." An agent orienting in the system reads the small stable apex before any wide detail, so it inherits the invariants before it touches anything that might have drifted from them.
3. **Single owner per fact.** Each structural fact lives in exactly one optic. The apex holds *only* enduring intent and explicitly pushes detail down: "Detail lives in the optics (single owner per fact); this file holds only the enduring intent." Single-ownership is what prevents contradictions from multiplying — a fact stated in two places is a fact that will eventually disagree with itself.

The clearest recent proof that this governance is live is Pillar 1 itself: the death-of-fixed-thresholds principle was ratified *as* design-principle #19 with a single owner and an apex corollary (§2.4). A methodology claim from this very paper became a governed, single-owner law in the pyramid it describes — the pyramid eating its own argument is the strongest demonstration that it is a governance structure and not a filing cabinet.

3.3 Why significance-ordering beats flat documentation

Flat documentation — a pile of equally-weighted files, or one enormous file — fails an agent in three ways the pyramid fixes.

It has no read order. Faced with two dozen equal files, an agent either reads all of them (expensive, and Pillar 3 calls that the cardinal sin) or guesses. The pyramid gives a canonical read order: apex first for invariants, then the *one* optic that owns the question at hand. The index maps a question class to its owning optic, so the agent loads only what answers its current question and skips the rest.

It has no conflict-resolution rule. In flat docs, when file A and file B disagree, there is no principled winner — you get a debate or a coin flip. The pyramid's strict derivation gives a total order: whichever is higher wins, full stop. Conflicts become resolvable by rank instead of by argument.

It rots uniformly. Flat docs rot everywhere at once because everything is equally load-bearing and equally neglected. The pyramid concentrates stability where it matters: the apex is small enough to keep correct and is the most-read file, so it stays true; the churning base is *expected* to change and is regenerated from the level above. Rot is pushed down to where it is cheap to fix and away from where it would be catastrophic.

The slimming of this project's own design bible is the worked example: a 3,085-line monolith was reduced to roughly 1,500 lines of pure narrative, with structural facts redistributed to per-optic files under single-ownership and the base principles extracted into a roughly 180-line apex. The monolith was flat documentation; the result is a pyramid. The reduction was not deletion — it was *significance-sorting*: enduring intent rose to the apex, structural facts settled into their owning optics, and the narrative kept only what is narrative.

3.4 The pyramid as live infrastructure

This is not aspirational. The apex file exists, is marked as outranking every optic, and is ratified with architect sign-off. The index names it the constitution and orders the read. The optics are real files — `topology.md`, `probes.md`, `flows.md`, `lifecycles.md`, `config-shape.md`, `failures.md`, `design-principles.md`, and the rest — each owning one fact class. Many carry an executable `verify:` block in their frontmatter, and a single command (`bible:invariants`, a real script in the project's package manifest) runs them all as the merge gate. That is how the *derives-from-and-does-not-contradict* rule is mechanically enforced rather than merely asserted: an executable invariant turns a contradiction into a failed check, not a judgment call. The pyramid is load-bearing infrastructure with a constitution, a map, single-owner facts, and a gate.

4. Pillar 3 — Cheap Code-Traversal as the Basis of Next-Generation Vibe Programming

4.1 The claim

"Vibe programming" — steering a capable model through a codebase by intent rather than by writing every line — lives or dies on one quantity: **how many tokens the agent spends understanding the system before it can change it.** Every token spent re-reading a file to recall what it does is a token not spent thinking about the change. The naive agent re-reads the repository on every task, paying the full comprehension cost again and again, and its effective intelligence is whatever is left of its context budget after that tax. The claim of this pillar is blunt:

> The real foundation of good vibe programming is not a smarter model. It is *cheap traversal* — the plugins and structures that let an agent navigate code and knowledge for a fraction of the tokens, so the budget goes to thinking rather than to re-reading.

Cheap traversal is therefore a **first-class engineering investment**, on par with the features it serves — not a nicety to add later. A team that builds traversal infrastructure makes every future agent-task cheaper and smarter; a team that skips it pays the comprehension tax forever, on every turn, in the currency that most constrains the agent.

The comprehension tax has two halves, and it is worth naming them before the instruments, because the instruments split cleanly across them. The first is **reach cost** — the tokens spent *getting to* the right knowledge or state: finding the file, the fact, the tool, the runtime value. The second is **carry cost** — the tokens that the bytes you pulled keep costing on every subsequent turn they remain in the window: a tool output, a file dump, a JSON blob, a diff. A complete traversal investment attacks both. Most of this project's instruments attack reach; the sixth, added below, attacks carry, and an external system occupies that ground as direct prior art.

4.2 The instruments of cheap traversal in this project

This harness already treats traversal as infrastructure. Six instruments, each a way to reach — or to carry — the right knowledge or code for far fewer tokens than reading or holding the source would cost.

(a) The design optics as a pre-compressed map. The single largest reach-cost win is the design bible itself. Instead of re-deriving "how does the tool loop diverge in the bridge layer?" by reading the implementation, an agent reads `tool-loop.md` — the structural fact, pre-compressed, single-owner, already verified. The pyramid (Pillar 2) is not only a governance structure; it is a *traversal accelerator*. The index routes a question to its owning optic so the

agent loads one small file instead of grepping the tree. Loading the whole bible directory would defeat the purpose — the discipline is "only load the optic that answers your current question." A map you must read end-to-end is not a map; it is more territory.

(b) A knowledge graph. Prose optics are cheap for questions whose owning file you can name. For questions that cut *across* files — "what depends on the recipe matcher?", "what else breaks if I change this RPC?" — a knowledge graph is the cheaper surface, because it makes relationships first-class and queryable instead of forcing a multi-file read to reconstruct them. A graphify-style tool turns code and docs into a queryable graph; a traversal that would cost many file-reads becomes a single graph query. The graph is the cross-cutting complement to the optics' per-file compression.

(c) Deferred-tool loading. An agent that loads every tool's full schema up front pays a large fixed context cost before it does anything, and most of those schemas are irrelevant to the turn. Deferred loading inverts this: tools are present by *name* only, and their full schemas are fetched on demand via a search step, so the agent pays for a tool's definition exactly when it decides to use it. This is salience applied to the *tool surface* itself — relevance (which schemas to load) is computed live from the turn, not fixed at session start. A pre-loaded tool list is a fixed list; deferred loading makes it live. This instrument now has external, quantified validation: see §4.2(f) and the marketingskills result below.

(d) Paired read surfaces (probes). A first principle requires that "every write surface has a paired read surface returning the same state," and a dedicated optic owns the inspection primitives. For traversal, the payoff is that the agent never has to *infer* runtime state by reading the code that produces it — it reads the state directly through the probe. A paired read surface is a traversal shortcut: instead of "read the producer, simulate it in your head, guess the state," it is "ask the probe, get the state." This probe symmetry is a traversal economy as much as an observability one.

(e) The recipe matcher as an intent router. The cheapest possible traversal of the *behavioral* library is to not search it at all — to have the right playbook already selected before the first token. A recipe matcher does exactly this: it scores the prompt against recipe metadata locally, in microseconds, and seeds the matching plan at turn start (the seam is `seedPlanFromPrompt` in `recipe-matcher.ts`). The agent does not read the recipe library to find the right recipe; the matcher routes intent to the right playbook for free. The foundational criteria require this: "Recipes are matched to intent before every execution, however trivial it seems." Intent-routing is traversal of the work-graph reduced to a local scoring pass. Because matching happens code-side and only the *winning* plan is injected, the recipe catalog is uncapped by design — adding playbooks costs nothing at the context layer.

(f) Reversible compression of carried context. The five instruments above all reduce *reach* cost. They do nothing for *carry* cost: once the agent has pulled a tool output or a file dump into the window, it pays for those bytes on every subsequent turn until they fall out. The instrument that attacks this is **reversible compression with a retrieve path** — shrink what was pulled, keep the original cached, and restore it losslessly on demand. This is the runtime-context analogue of the never-delete-archive reversibility that Pillar 1 leans on: the §4.3 unification "observability doubles as traversal" has a natural sibling here — *compression with a retrieve path doubles as eviction*, because a compressed-but-restorable artifact is exactly an evicted-but-recoverable one. In this project the lineage is the total-recall soft-delete invariant (nothing is ever destroyed; pointers compact, originals survive); the carry-cost instrument is that same invariant applied to the live window rather than to the recipe and skill library. The external system that has productized precisely this gap is treated as first-class related work in §4.3 — naming it honestly is part of the contribution, because it ships the measurement this paper otherwise lacks.

4.3 Why cheap traversal is an investment — and the external systems that prove it

The case for treating traversal as first-class rests on a compounding argument, two ties to first principles, and — new in this revision — two external open-source systems that validate the posture at scale and, by doing so, expose a real weakness in this paper that the revision must answer honestly.

The compounding argument: traversal cost is paid *per turn, forever*. A one-time investment in a pre-compressed map, a knowledge graph, deferred loading, probes, an intent router, and reversible compression lowers the per-turn comprehension tax on *every subsequent turn*. Over a deployment's life that is the difference between an agent whose budget mostly goes to thinking and one whose budget mostly goes to re-reading. The investment amortizes to near-zero per turn; the tax never does.

The tie to **observability**: a paired read surface is simultaneously an observability instrument and a traversal instrument. The same probe that lets a human watch live state lets the agent read that state cheaply instead of inferring it. Cheap traversal and total observability are not two investments; they are one — every read surface you build for the human is a read surface the agent traverses for free.

The tie to the **correlated artifact chain**: when every artifact — a recipe run, a subagent dispatch, a memory write, a plan step — carries the correlation id that links it to its neighbors, traversal of the *execution* graph becomes following links rather than reconstructing the chain by reading logs. The artifact chain is the runtime analogue of the knowledge graph: it makes "what happened, and what did it cause?" a link-following query instead of an expensive forensic read.

Related work — reversible compression at carry-time: chopratejas/headroom. The most direct external prior art for this pillar's central thesis is **headroom** (chopratejas, Apache-2.0, ~24.7k). Headroom occupies exactly the carry-cost half of the comprehension tax that this paper's five original instruments leave untouched. Its reversible **CCR (Compress-Cache-Retrieve)** keeps originals cached and lets the model retrieve on demand — convergent with this project's own eviction and pointer-compaction lineage (the total-recall soft-delete invariant), but applied to *live context* rather than to the recipe and skill library. It adds per-content-type compressors: statistical JSON-array crushing (a "SmartCrusher"), AST-aware code compression over tree-sitter, and dedicated log, diff, and text handlers. It claims 60–95% token savings with roughly zero accuracy delta on GSM8K, TruthfulQA, SQuAD, and BFCL, shipped as a reproducible eval suite (`python -m headroom.evals`), with a trained compression model (Kompress-v2-base) and proxy, MCP, and library form factors.

The honest differentiation cuts three ways, and one of them is a weakness this citation deliberately exposes. (i) **headroom is the sixth instrument, and it is orthogonal to the other five.** Those five reduce *reach* cost — find the right file, state, or tool fast. Reversible CCR reduces *carry* cost — shrink what you already pulled and restore it on demand. Pre-compression at the source and reversible compression at carry-time are complementary, not competing: this project never pulls the byte in the first place (single-owner optics, knowledge graph, intent router), while headroom shrinks the byte once it is pulled. A complete system wants both, and the paper says so. (ii) **headroom validates the first-class-investment claim from the outside.** §4.1 argues that traversal-cost reduction deserves to be a productized layer; headroom *is* that layer, built by an external team, with proxy, MCP, and library form factors and a measured savings-versus-accuracy curve. The posture this pillar argues for is not idiosyncratic — someone else productized it at scale. (iii) **headroom exposes the measurement gap in this paper, and the revision concedes it.** Headroom ships *numbers*; Pillar 3 ships none. The compounding argument ("traversal cost is paid per turn, forever") is asserted, never measured — there is **no quantified token-savings claim anywhere in this paper.** That is a real weakness, and headroom's reproducible eval frame is the template for fixing it: a per-content-type savings table with an accuracy-delta column on a fixed public benchmark would

turn "cheap traversal compounds" into a curve. What this project does that headroom does not: significance-ordering (Pillar 2) means the map is *governed* — single-owner, non-contradiction, executable `verify:` — not merely compressed; headroom compresses arbitrary content with no constitution above it.

Related work — deferred loading, measured externally: the marketingskills router-eviction. The deferred-loading claim of §4.2(c) and the intent-router claim of §4.2(e) were, until this revision, argued only from this project's own internals. There is now a clean, quantified, *external* instance of the same mechanism. Applying total-recall-style eviction to **coreyhaines31/marketingskills** (~33k, a 44-skill marketing catalog), **one ~130-token router skill replaced ~915 tokens of always-loaded skill descriptions, with all 44 frameworks still reachable on demand** — a roughly 7× reduction in always-resident catalog overhead on a third-party library, achieved by exactly the move §4.2(c)/(e) describe: replace a pre-loaded list with a code-side router that injects only the matched unit.

This is the first concrete before/after *number* this pillar's deferred-loading claim has ever had (915 → 130 tokens), on a real, named, large external catalog — the reach-cost counterpart to the carry-cost numbers headroom supplies. It also ties Pillars 1 and 3 together at a measurable seam: an always-loaded description block is *simultaneously* a fixed list (Pillar 1 — it freezes "these are the skills" and pays for all of them every turn) and a traversal tax (Pillar 3 — the agent carries ~915 tokens it mostly does not use). The router retires both at once: uncapped catalog (add frameworks freely, code-side matching scales), live relevance (only the matched skill's body is injected), near-zero resident cost. The honest differentiation to record is that marketingskills ships two disciplines this project's skill stdlib does **not** yet have — a per-skill `evals/evals.json` (realistic prompts plus assertion lists, runnable by an LLM judge) and a shared context-contract file every skill reads before asking the user questions. Those are exactly the eval and shared-context mechanisms a Pillar-3 catalog needs to *prove* the router preserves reachability: an evals.json per recipe is the missing test that a 130-token router still reaches all 44 frameworks correctly. The router is the mechanism; the evals are the proof obligation, and this paper currently has the former without the latter.

So cheap traversal is not a convenience bolted onto the system. It is the precondition that makes Pillar 1 affordable — live computation requires reading the live situation cheaply — and makes Pillar 2 useful: a pyramid no one can traverse cheaply is just a tall pile. The two external systems above are the evidence that the posture is sound and the honest mirror that shows where this paper still owes a number.

5. The Embodiment — A Native Orchestration Runtime (Planned; Its Derived-Bound Substrate Now Partly Live)

5.1 Why one runtime embodies all three pillars

The three pillars are a methodology. Their executable embodiment is a single piece of planned infrastructure: a **native orchestration runtime** inside the agent's own gateway, exposing four primitives — `agent()` (spawn one worker), `parallel()` (fan out with a barrier), `pipeline()` (per-item stages without a barrier), and `phase()` (a labeled stage surfaced to the observability panel) — over substrate that already exists: the subagent spawn path, the recipe-runner's group scheduler, and the `onRecipeState` sink (both real seams today, in `recipe-runner.ts` and `recipe-rpcs.ts`). It is governed by this paper.

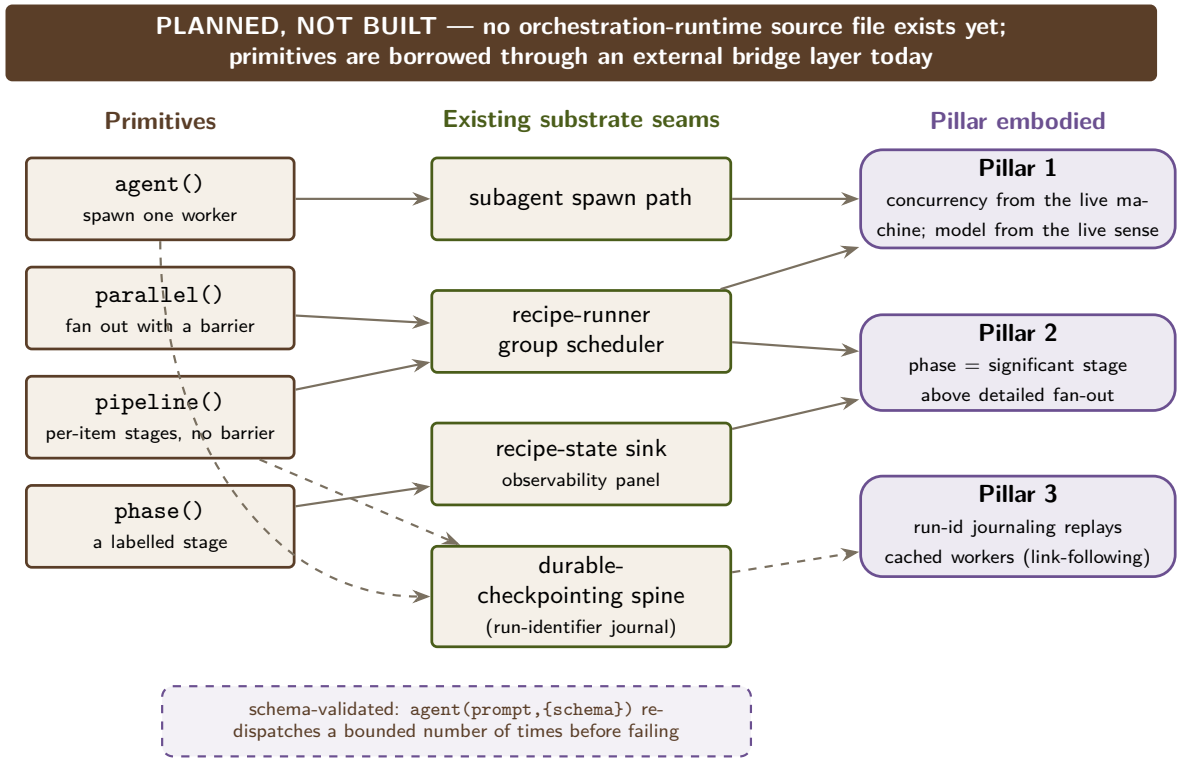

Figure 2. The orchestration runtime mapping each of its four primitives onto an existing substrate seam and onto the pillar it embodies: agent and parallel draw concurrency from the live machine (Pillar 1), phase surfaces stage shape above detailed work through the recipe-state sink (Pillar 2), and runId journaling replays cached agents so resume is link-following over a durable spine (Pillar 3).

It embodies the three pillars at once:

- **Pillar 1 (no fixed limits).** Concurrency is derived from the live machine (`min(16, cores - 2)`), not a frozen cap; budget for a fan-out is reasoned against the real remaining allowance and the value of the work, not a hardcoded ceiling; the model each agent runs is chosen from the live model sense, not a pinned ID. This is no longer purely a design intent. The first family of situation-derived bounds now runs *at real call sites* inside the recipe-runner and subagent substrate the runtime is specified over: **`deriveSpawnBudget`** (parallel-spawn fan-out *width* — rises with the typed-field surface, gains a unit when a skill backs the step, widens as fitness falls so a shaky recipe spawns wider coverage, clamped down by affordability, floored at 1); **`deriveRecoveryRetryBudget`** (how many times the runner re-runs a failed step under `onError: retry` — scales with uncertainty `1 fitnessSuccessRate`, decays with prior attempts, affordability-clamped); **`deriveOverseerLoopBudget`** (how many supervision passes the Overseer makes — rises as fitness falls, gains a pass while the gap-to-done is still shrinking, decays with prior iterations); and a companion **`redispatch-budget`** for schema-retry attempts. Each is a pure function whose output responds to live inputs, unit-tested so that more fields or lower fitness yields a strictly larger bound — never collapsible to a literal without re-introducing the frozen MAX the paper condemns. The author's directive (`max-tokens`, `onError: retry N`, fan-out width) is treated as a *downward* cap over the derived bound (`min(authorIntent, derived)`), and a missing or `{{template}}` directive **fails closed to the derived bound, never throws** — the salience posture (compute live; degrade, don't abort) made mechanical. The runtime is, by construction, a place where the death-of-fixed-thresholds discipline is enforced rather than merely recommended — and at these seams it now is.
- **Pillar 2 (significance-ordered fan-out).** An orchestration script is itself a small pyramid:

a `phase()` is the significant, stable shape of a stage; the `parallel()`/`pipeline()` fan-out under it is the wide, detailed work; the per-agent prompts are the churning base. Phases surface to the observability panel through the same `onRecipeState` sink the recipe runner uses, so the significant structure is visible above the detailed work.

- **Pillar 3 (cheap traversal of the work-graph).** The runtime journals phases and per-agent results keyed by a `runId`, so resuming a run replays cached completed agents and only re-runs the pending ones — traversal of the work-graph as link-following rather than re-execution. This is the execution-time analogue of the knowledge-graph traversal in Pillar 3, over a durable-checkpointing spine that already exists.

It is also schema-validated: `agent(prompt, {schema})` validates the worker's JSON output against a JSON-Schema and re-dispatches a bounded number of times before failing — structured output as a first-class primitive, not a hope. The re-dispatch bound is no longer the "fixed default of two attempts" v1.1 footnoted as "worth a review pass before it ships": it is now itself a derived bound (`redispatch-budget`), so that honesty hedge is retired.

The derived bounds read from a measured input, not an assumed one. Every derivation above takes `fitnessSuccessRate` and widens coverage when it is low. That input is now closed end-to-end by an **empirical-fitness loop**: a producer (`stampRecipeAttribution`) appends a `[recipe_attribution] recipe:<owner/slug>` marker into the engram event store on every run (a `turnId:0` system event that cannot mis-split an episode), and a consumer (`makeFitnessLookup(...)(kitRef)`) reads the measured success rate back and threads it into the `derive*` calls. A matcher fitness-key bug — keying the lookup on something other than the canonical `owner/slug` — was fixed so producer and consumer agree. The live computation's confidence term is now *measured fitness*, not a `0.5` default: Pillar 1's input is observed, not guessed.

The canonical worked example of frozen-to-derived is the **Overseer** (the "is the whole task done?" critic). Its live per-turn engine previously used a frozen `MAX_OVERSEER_ITERATIONS = 5` — a textbook #19 violation sitting in the shipped, per-turn path. It now computes `min(OVERSEER_LOOP_HARD_CEILING = 25, deriveOverseerLoopBudget({fitnessSuccessRate, gapShrinking}))` re-derived each turn; the frozen `5` is gone, and `25` survives only as the structural *ceiling* the law permits, never the working value. Both overseer paths (the recipe-runner loop and the per-turn engine) now share one derivation source, so they cannot drift. Because the rebuilt Overseer is a BROCA recipe with a right-side "keep-going" nudge bubble wired through `prefrontal.recipe.run`, its derived loop surfaces its live state on the observability panel — a Pillar-3 observability-as-traversal instance for the derived bound itself: a human, or the agent's reflection layer, watches the loop reason in real time. "Observability is seeing the fall coming," met for the bound the runtime derives.

A further family of derived bounds landed on the recipe missing-value resolution path — `deriveContextTimeoutMs` (wait for a structured extraction of a missing recipe var from the live conversation), `deriveMemoryTimeoutMs` (wait for an engram recall of a missing var), and `deriveAskTimeoutMs` (the human-answer wait for the durable ask-for-missing pause, floored at a realistic human turn-around) — each scaling with the count of missing vars and `1  fitness` uncertainty, each unit-tested to respond to its inputs, each bounding a *best-effort* tier that falls through on expiry rather than hard-stopping. A derived timeout that degrades gracefully is the precise opposite of a frozen cliff, and these are its value-resolution members.

5.2 Honest status — planned, governed, with the derived-bound substrate now live

The four orchestration primitives are still **planned, not built.** `agent()/parallel()/pipeline()/phase()` are **not defined in this project's source** (no `orchestration-runtime.ts` exists yet); today they are a borrowed workflow tool reached through the bridge layer, alive only while that

external runtime is driving the turn. What *has* moved since v1.1 is the substrate beneath them: the situation-derived concurrency, retry, supervision, and fan-out bounds the runtime was *specified* to compute now run at verified seams (§5.1), and an earlier increment landed the recipe-runner skill substrate (`invoke skill:`, semantic `fork.skill.search`, deterministic `prefrontal.recipe.compose`, a live-margin promote gate) end-to-end on a real call site. The strategic finding behind the implementation plan still holds: the agent already *has* most of the capabilities — real cross-provider debate, tree-of-thought, curiosity, an overseer, recipe fitness and evolution, deferred tools, hooks, plan mode, memory, schema-validated outputs — but it has many of them *inside borrowed workflows*. The plan makes the primitives native so the agent stays fully capable through the near-term programmatic-metering change and CLI pin — a deadline that is itself a cautionary tale for Pillar 1, because a hardcoded dependence on a borrowed runtime is a fixed artifact whose world is about to move.

Two honesty caveats are carried forward in the paper's own no-silent-failure voice, because the substrate that is live is not uniformly live: the `budget-exceeded` arm in `orchestration-deps.ts` is **wired-but-inert** until `agent.wait` carries the exhaustion reason — recorded PARTIAL exactly as it is commented in source — and the affordability clamps in the `derive*` family are **structurally-derived-but-inert** until a caller threads a real remaining-token budget — also PARTIAL. Recording these honestly is not a hedge; it is the application of the *no silent failure* principle to the paper itself. A producer is "wired" only when verified firing at a real call site — a dead producer behind green tests is the failure class most distrusted, and a paper that claimed a planned mechanism as shipped would be the documentation form of that exact failure. A companion change-ledger tracks each piece against a designed / partial / live status, flipping entries to live with a commit SHA only as they land at a verified call site.

6. Situating the Work

This paper is the *methodology* layer of a larger body of work: it argues the principles, while a separate execution substrate and a separate theory of abstractions build and explain the machinery. To keep this paper self-contained, the borrowed ideas are summarized where they are used rather than assumed. The external open-source systems below are treated as first-class related work, not footnotes — they are fresher than any static citation and, in two cases, supply the measurement this paper's own claims lack.

The ratified base principles. A short, high-altitude constitution sits above this paper and above every structural optic. It states, among others: capability-first with prudence as the brake, total observability through a correlated artifact chain, a paired read for every write, recipes that evolve under reversibility, the fractal budget doctrine, single-owner-per-fact, and — ratified this cycle — the derive-bounds-from-the-live-situation law (design-principle #19) that Pillar 1 expands. Where this paper and the constitution could appear to differ, the constitution is right by construction and the discrepancy is a bug in this paper.

The recipe execution substrate. A separate body of work owns the executive-function machinery: the recipe matcher, recipe composition, on-the-fly authoring, loops, an effort router that picks a model and a thinking budget from live signals, the observability panel, and durable checkpointing. A *recipe* there is a named, matchable playbook — a unit of behavior the agent selects and runs. This paper borrows three things from it: the matcher as the intent router of §4.2(e), the effort router as a live-computation instance of Pillar 1, and the durable checkpointing spine the orchestration runtime resumes over. This paper does not redefine the recipe; it treats it as already defined.

The theory of intermediate abstractions. A further body of work argues that the recipe is the missing middle layer between a bare wish and a hand-written script, and identifies a set of reusable primitives. Three of them are load-bearing here: a *multi-optic atlas* (the pre-compressed

documentation map that *is* the traversal accelerator of Pillar 3); *executable invariants* (the `verify:` blocks that mechanically enforce the pyramid's non-contradiction rule, Pillar 2); and *probe symmetry* (the paired read surfaces that double as a traversal economy, §4.2(d)). The same body of work supplies the discipline of honest claims-softening that §5.2 follows. Where it argues the recipe is the right *unit*, this paper argues that *relevance over that unit must be computed live*, and that *the units must be ordered by significance and cheap to traverse*.

External system — reversible context compression (chopratejas/headroom). Positioned in full in §4.3: the most direct external prior art for Pillar 3, productizing reversible Compress-Cache-Retrieve over live context with content-type-specific compressors, public-benchmark savings numbers, and proxy/MCP/library form factors. It is complementary to this project (compress-at-carry versus pre-compress-at-source), it validates the first-class-investment posture from the outside, and it honestly exposes the absence of any quantified token-savings number in this paper.

External system — measured deferred loading (coreyhaines31/marketingskills). Positioned in §4.3: a 44-skill catalog on which a ~130-token router replaced ~915 tokens of always-loaded descriptions with all frameworks still reachable — the first external before/after number for the deferred-loading and intent-router claims of §4.2(c)/(e). It also models two disciplines this project's skill stdlib lacks: a per-skill `evals/evals.json` and a shared context-contract file. Those are named here as the proof obligations a Pillar-3 catalog should adopt to *demonstrate* — not merely assert — that its router preserves reachability.

External systems — authoring discipline and workflow distribution (addyosmani/agent-skills; the Journey registry). Two further fresh external surfaces bear on Pillar 2's governance and the recipe-as-unit lineage, and honesty requires positioning them precisely rather than overclaiming. **addyosmani/agent-skills** (MIT, ~56.8k) packages engineering-discipline skills, subagent personas, and command definitions in a single plugin; roughly seventy percent of it overlaps this project's own discipline stack, so it is *validating* rather than novel for the most part. Two of its authoring conventions are genuinely worth adopting and sharpen Pillar 2's authoring side: an explicit **"When NOT to use"** section per skill and a **"Loading Constraints"** section declaring *where* a skill may load (main session versus subagent). Both are significance-ordering applied to the *authoring* of a unit — they pin, at the apex of each skill, the conditions under which the unit is relevant, which is the Pillar-1 "compute relevance, don't assume it" discipline pushed up into documentation. The **Journey registry** is a distribution layer for real, runnable agent workflows (including third-party repacks of suites such as the one above); it is relevant to this paper only as evidence that the *unit* the recipe substrate formalizes — a matchable, reusable, shareable playbook — is becoming an externally-traded artifact, which raises the stakes on the evals-and-shared-context proof disciplines §4.3 borrows from marketingskills. Neither system changes a claim in this paper; both are cited as the live external context the methodology now sits inside.

7. Status and Future Work

7.1 What is real today

- The **pyramid (Pillar 2)** is live infrastructure: the apex file is ratified and authoritative, the index routes by significance, optics own facts singly, and a single invariant-runner command enforces non-contradiction via executable `verify:` blocks. Pillar 1's own principle is now ratified *inside* this pyramid as single-owner design-principle #19 with an apex corollary.
- Several **cheap-traversal instruments (Pillar 3)** are live: the optics as a pre-compressed map, a knowledge graph for cross-cutting queries, deferred-tool loading (with an external 915→130-token before/after on the marketingskills catalog), paired read surfaces, and the

recipe matcher as an intent router. The correlated-artifact-chain and paired-read-surface requirements are first principles already enforced in the substrate. Reversible carry-time compression is named as the sixth instrument and positioned against headroom's external implementation; it is not yet built in this project.

- **Pillar 1 is no longer "argued plus four examples."** It is a ratified design law (#19) with a categorical-vs-quantity typology, a 17-instance whole-system audit (with the deliberately-hard categorical exceptions named), and a derive→validate→enforce→catch safety chain. The first family of situation-derived bounds — `deriveSpawnBudget`, `deriveRecoveryRetryBu` `deriveOverseerLoopBudget`, the schema `redispatch-budget`, and the value-resolution timeouts — runs at real call sites, fed by a measured empirical-fitness loop. The live per-turn Overseer's frozen `5` is retired to a derived budget under a `25` ceiling. The `rank` leaderboard and the `<70%` budget gate remain legacy artifacts still in place in the live config, named here as the work, not claimed as replaced.

7.2 What is planned

- The **native orchestration runtime (the embodiment, §5)** is planned, governed by this paper, and specified task-by-task; the four primitives `agent()/parallel()/pipeline()/phase()` are not yet native, but the concurrency, retry, supervision, and fan-out bounds the runtime is specified to derive now run at verified seams. Remaining: the four primitives themselves plus an additive `recipe.orchestrate` RPC; runId journaling and resume; per-agent git-worktree isolation; a tool-RPC pipeline that collapses many tool round-trips into one follow; and flipping the two PARTIAL arms (the inert `budget-exceeded` reason and the inert affordability clamps) to live by threading the exhaustion reason and a real remaining-token budget.
- The **live model sense** that retires the `rank` leaderboard, and the **fractal budget computation** that retires the `<70%` gate, are the two highest-leverage Pillar-1 replacements still to ship — each converting a named legacy artifact into a live computation, each made safe by reversibility (a kill-switch back to the static value) and observability (surfacing the live inputs it reasoned over).
- **Measurement is the named debt.** Pillar 3 ships no quantified token-savings number of its own; the revision concedes this (§4.3) and adopts headroom's reproducible-eval frame — a per-content-type savings table with an accuracy-delta column on a fixed benchmark — and marketingskills' per-unit `evals.json` discipline as the proof obligation that the optics map, the knowledge graph, and the intent router actually compound and actually preserve reachability. The 17-cliff audit is the standing Pillar-1 backlog, with the fix shape (ceiling-not-working-value + derivation formula + adaptive pressure) already defined.

7.3 The honest boundary

The methodology — all three pillars — is articulated and largely instantiated, and Pillar 1 has hardened from an argued principle into a ratified, audited, partly-shipped law. The embodiment's four primitives are designed and not yet native, while the derived-bound substrate beneath them is live at real seams. Drawing that line precisely is the discipline the paper argues for, turned on itself: a fixed claim of "done" against a moving implementation would be exactly the stale-artifact anti-pattern Pillar 1 condemns. This status section is therefore *itself* a live computation — it reports what is true on the date of writing and is expected to be re-derived, not frozen, as the runtime lands.

8. Conclusion

The salience network is the brain's standing refusal to decide relevance in advance. It re-computes, every moment, what matters now — and that refusal to freeze is precisely why a brain stays adaptive in a world that will not hold still. This paper has argued that an agent harness needs the same refusal, and that it decomposes into three pillars. *Kill the fixed thresholds and fixed lists*, because each encodes a dead world-state and drifts the system silently into obsolescence; replace them with live computation, made safe by a derive→validate→enforce→catch chain rather than by the constant's rigidity, and grounded in a stated trust model — a single trusted principal gains friction, not safety, from a quantity gate. *Order knowledge and code as a pyramid of significance*, so that "everything is live" does not mean "anything goes" — a small stable apex pins the invariants the churning base inherits, contradictions become typed bugs resolved by rank, and the law that governs Pillar 1 now lives *inside* that pyramid as a single-owner principle. *Invest in cheap traversal as a first-class concern*, attacking both the reach cost of getting to knowledge and the carry cost of holding it — and measure it, because the external systems that productized this posture (headroom's reversible compression, the marketingskills router) ship the numbers this paper still owes.

The embodiment that fuses all three — a native orchestration runtime with no frozen limits, significance-ordered fan-out, and cheap work-graph traversal — has its derived-bound substrate live at real seams and its four primitives still designed, named here honestly as not yet native. That honesty is not incidental to the thesis; it *is* the thesis. A fixed claim against a moving system is the stale artifact the whole paper is against.

The programmer who hardcodes a threshold is encoding yesterday's world into tomorrow's system. The one who computes relevance live, orders it by significance, and makes it cheap to traverse — both to reach and to carry — is building a system that stays as smart as the world it runs in. That is not better programming. It is salience.

Appendix A — The Fixed-Artifact Test (Decision Procedure)

```
function CLASSIFY_BOUND(constant):
    # Design-principle #19: categorical boundaries stay hard; quantity thresholds derive.
    if constant is a CATEGORICAL capability/permission boundary
       (may/may-not do X, API-required field, PII split, security gate, tool whitelist):
        return HARD          # keep: a category, not a quantity - freezing it is correct
    if constant is a QUANTITY threshold (at most N, after T seconds, up to D deep, fan W):
        if the quantity MOVES in the deployment's lifetime:
            return DERIVE    # replace working value with a live computation; keep the
                             # frozen number only as a CEILING, never the working value
        if it reads a genuinely-fixed physical fact (core count, page size):
            return PHYSICAL  # keep; reading it at runtime is already "live"
    return REVIEW            # default suspicion: most magic numbers are DERIVE

function REPLACE(quantity_bound):
    working <- derive_from_live_situation(quantity_bound.domain) # responds to live inputs
    working <- min(working, ceiling)                  # frozen number survives only here
    working <- max(working, 1)                        # floored, never zero
    on_approach(ceiling): emit_adaptive_pressure()    # cliff -> ladder: warn, don't cut
    on_exhaustion:        persist_partial(); warn(); continue   # degrade, don't abort
    validate(working)                                 # reject malformed at the boundary
    enforce(working)                                  # runtime watchdog, not advisory
```

```
    expose(working.inputs, working.result)            # observability: show the reasoning
    guard <- kill_switch_back_to(frozen_value)        # reversibility: undoable
    return working  # behind guard, validated, enforced, catchable, derived from now
```

Appendix B — The Three Pillars on One Page

Pillar	The anti-pattern it kills	The salience replacement	Safety mechanism	Principle anchor
1. Death of fixed thresholds/lists	hardcoded number, frozen ranking, round-number cap	live computation over the moment-of-use situation; frozen number survives only as a ceiling	derive→validate→enforce→catch; reversibility + observability; single-trusted-principal trust model	design principle #19; budget & autonomy doctrines
2. Pyramid of significance	flat docs: no read order, no conflict rule, uniform rot	apex → optics → derived principles → diagrams → code; derive-don't-contradict	executable `verify:` invariants; single owner per fact; apex outranks	the pyramid section; #19 ratified inside it
3. Cheap traversal	re-reading the repo every turn (reach tax) + carrying pulled bytes every turn (carry tax)	optics map + knowledge graph + deferred tools + probes + intent router (reach) + reversible CCR (carry)	observability doubles as traversal; correlated artifact chain; per-unit evals as reachability proof	total observability + artifact chain; headroom & marketingskills as external evidence
Embodiment (substrate live, primitives planned)	borrowed runtime alive only under the bridge layer	native `agent/parallel/pipeline/phase`; schema-validated, live concurrency/budget; derived bounds already at real seams	additive RPC behind flag; derive→validate→enforce→catch; PARTIAL arms recorded honestly	all of the above; capability-first;

References