Privacy-as-Load-Bearing, and the Recipe as Intermediate Abstraction

Oscar Serra (Independent Research) · Architect, Claude Opus 4.7 (TinkerClaw, anchored on OpenClaw)

2 June 2026

Abstract

This paper carries two arguments. The first is structural: a privacy gate scoped to each push proves that no single push makes history worse, but it does not prove the cumulative history is clean — a diff-based contract is necessary but not sufficient for any contract that must hold over absolute state. A real leak demonstrated this. The fix names the missing axis (privacy dominates functionality) as a design principle and hardens the pre-push gate to scan accumulated drift, not just the push range. The second argument is about abstraction: AI-pair coding lacks a *middle* layer between source code and a single prose mental model, and that gap exists for behavior as much as for documentation. The fork ships an orchestration substrate of *recipes* (on disk: *kits*) — the named, replayable, gated unit between a one-shot prompt (a wish) and a script (brittle code). The same five primitives that make a self-documenting codebase enforce itself make the recipe library enforce itself. Two claims are deliberately held to what the live code supports: the composition algebra has both halves shipped and tested — `uses:` (runtime sub-recipe invocation) and `composes:` (inline step-merge at plan-build time), each depth- and cycle-guarded; and recipe matching is lexical by default with a semantic fallback lane gated default-OFF, not a default-semantic match. A third position is now defensible against the live ecosystem: the recipe's on-disk schema (`kit/1.0`) is a *cross-registry interchange format*, not a fork-local convention — the same header is published by an independent registry (Journey) and parses and runs in our loader, completing the function/module-library analogy across project boundaries. The honest residue is also named: every gate the recipe layer carries is *structural*, and adjacent open-source systems (addyosmani/agent-skills, marketingskills, headroom) carry contracts we do not yet — negative-space anti-triggers, load-context constraints, and runnable behavioral evals — which the paper now positions as related work and queued future work rather than absorbing into an inflated maturity claim.

1. The accumulated-drift failure pattern

1.1 What happened

Between 2026-05-11 (when the pre-push PII gate shipped — commit `97f5baea08`) and 2026-05-12, eight Architect commits and one Jarvis commit landed on `origin/develop`. Each push

triggered the gate. Each run scanned `<remote_sha>..<local_sha>` for the PII regex on lines starting with `+`. Each run reported zero hits. Every push succeeded.

On 2026-05-13, when the user asked to advance `main`, the architect ran `git log -p origin/main..origin/develop` over the same regex. Result: 16 hits. None had appeared in any individual push range, because:

- **Type A — content unchanged across commits.** Lines like the per-chat-strategy design note lived in `bible.md` before the gate shipped. They carried forward verbatim when `bible.md` was slimmed into `wa-triggers.md`; the slim script left the line unchanged in the destination file. No push added a new PII line — the line was old.
- **Type B — content edited but PII preserved.** Lines that did change, but where the PII token survived the edit. Each push's diff showed a `-<old>` followed by `+<new>` where `<new>` still carried the matched token. The gate scanned `+` lines and flagged them, but the hit was either (a) inside a self-excluded file (the hook excluded `bible.md` at one point, then stopped after a restructure), or (b) inside an earlier develop commit that the per-push scan re-flagged while the user-visible test focused on the latest push.

The net result: develop accumulated PII across 50+ commits while every individual push passed its own gate.

1.2 The pattern in one sentence

> A diff-based gate scoped to the push range proves each push doesn't make things worse. It does not prove the cumulative state is clean.

1.3 Generalisation beyond PII

The pattern holds for any contract that must hold over the cumulative state rather than over each step:

- Schema-drift contracts: one migration may not violate the invariant; a chain of three may land in a state none of them individually broke.
- Performance contracts: one commit may not regress a benchmark by 5%; seven commits each regressing 0.9% land beyond the threshold.
- Bundle-size contracts: one dependency add may be small; the accumulated tree exceeds budget.

Anywhere a contract reads "the system state must satisfy X," a per-delta gate is necessary but not sufficient. The gate must check the absolute state, not just the delta.

2. The constraint that ranks the primitives: privacy dominates functionality

The discipline rests on five primitives:

1. Multi-optic atlas
2. Executable invariants
3. Probe symmetry
4. Commit-and-timestamp anchors
5. Manifest-injection round-trips


Per-Pus
Each pus

To these it adds a constraint that primitives 2 and 3 operate under:

> **Constraint P — Privacy dominates functionality.** When a verify command catches a privacy-contract violation, no functional work proceeds until the violation is fixed. The PII boundary is not graded against functional deadlines; it is a hard precondition for every push and every promotion. Bypasses exist (`PII_GUARD=off` for the maintainer's-own-byline case) but are emergency-only and require written justification.

This is the discipline's only explicit priority-axis constraint. The five primitives are otherwise co-equal — failure of any one means the gate refuses. The constraint acknowledges that privacy failures are not interchangeable with functional ones:

- A verify that fails because a probe returned the wrong shape: refusable, fixable next session, no permanent damage.
- A verify that fails because a PII pattern reached committed history: refusable, fixable next session, but the public history still records the violation, irreversibly.

Asymmetry of consequence demands asymmetry of priority. The constraint formalises this.

3. The design principle

The constraint is codified in `design-principles.md`:

> **17. Privacy precedes functionality.**
>
> **Why.** Privacy is not a co-equal axis with functionality; it dominates. A change that ships private data degrades trust in a way no later functional improvement can undo. The PII boundary is the contract; the leak-grep gate is the enforcement; both must run before main can advance and before the maintainer says "ship it." When sanitization is in tension with delivery speed, sanitization wins.
>
> **How to apply.** (a) Treat `PII_GUARD=off` as emergency-only; every use requires written justification. (b) The pre-push hook scans two scopes: the push range and `origin/main..HEAD`. (c) When auditing before a `develop` → `main` merge, run the canonical leak-grep over `origin/main..origin/develop` excluding self-referential files. Zero hits, no exceptions. (d) Educational files use `<FirstName>` / `<owner-e164>` placeholders so they don't trip their own grep.

The principle text is present in the live bible: `design-principles.md` carries a self-verify block asserting the string "Privacy precedes functionality" exists, citing the 2026-05-13 incident (the verify block sits at line 16; the heading at line 131 — both confirmed). The runtime enforcement is the gate-hardening below.

4. Gate hardening — accumulated-drift scope

`scripts/pii-pre-push.sh` scans two scopes on every push, both confirmed in the live script's scan loop:

1. **Push range:** `${remote_sha}..${local_sha}` for each pushed ref, scanned commit-by-commit (`git log -p`).

2. **Accumulated drift:** `origin/main..${local_sha}` for each pushed ref not already an ancestor of `origin/main` (the script skips the scope when the pushed snapshot is already on `origin/main`; the scope-2 block begins at line 126).

The two scopes deliberately use different scan modes. Scope 1 walks history commit-by-commit, so it flags a PII token in any commit being pushed. Scope 2 scans the *net diff* of the snapshot against `origin/main` — `git diff origin/main HEAD` semantics, so an add-then-remove cancels: a token introduced in commit A and removed in commit B inside develop's own history is not flagged, because the state main would inherit is clean. An earlier attempt scoped scope 2 with `git log -p` and false-positived on exactly those cleaned-up historical leaks; the switch to diff semantics is what makes scope 2 mean "what main would inherit" rather than "everything develop ever touched." Same regex, same exclusion list, same report — only the range and the mode differ.

Cost: one additional `git diff` per pushed ref against main. In practice ~50 commits of accumulated develop, sub-second. Bounded.

Bypass: the same `PII_GUARD=off`, applying to both scopes — there is no scope-1-only partial bypass. The bypass means "I have read the situation and am acknowledging," not "this scope is too noisy."

5. Failure-mode generalisation: who else has this shape?

Three contract classes already in the discipline, read through the cumulative-state lens:

- **Negative-evidence logging — same shape.** A per-call check that "the expected event fired within N seconds" is per-delta. The cumulative version is "across the operation's full graph, no expected event went unfired." The per-call completion-tracking wrapper is the per-delta version; a "every operation started in the last 24h has a matching `done` event" variant would be the accumulated-drift counterpart.
- **SLOs + burn-rate — already in this shape by design.** SLOs are absolute-state contracts (at least 95% over a window); the burn-rate computation evaluates the window, not each event. SLOs were the example the discipline had right by accident.
- **Bug-pattern linter — currently per-delta.** It scans the tree, each rule firing on a local pattern. A rule could instead check the cumulative count of pattern instances, refusing the PR that introduces the Nth. Future enhancement.

The lesson: every contract should be examined for "is the per-delta check sufficient, or do we need a cumulative-state check?" The answer is "cumulative" more often than "per-delta," but cumulative checks cost more to implement, so per-delta is the default. The discipline now flags that default as a known shortcut, not a complete solution.

6. Memory and agent updates

The principle is durable across sessions only if both agents — the Architect (Claude Code in `~/src/jarvis-icu`) and the Jarvis cc-bridge runtime in `~/.openclaw/workspace/` — reach for it unprompted. Three updates landed alongside the principle:

1. **Architect-side auto-memory** at `~/.claude/projects/-home-globalcaos-src-jarvis-icu/memory/` — the WHEN/HOW (sanitization-first sequencing, canonical command, "do not say 'pushed' until zero hits"). Registered at the top of `MEMORY.md`.

2. **Jarvis-side workspace** at `~/.openclaw/workspace/AGENTS.md` — a bullet under Engineering Discipline: "Sanitization before push," plus the canonical command. Visible to Jarvis on every session.
3. **Jarvis-side knowledge note** at `~/.openclaw/workspace/memory/knowledge/tinkerclaw-bible.md` — extends "Don't do these" with "Don't push to tinkerclaw without checking sanitization first."

The pre-push hook is the last line of defence; the memory entries make it the first thing the agents check unprompted.

7. What the maturity claim can honestly say

The blunt claim — "the discipline enforces itself; when a fork commit breaks one of its claims, the push is refused" — is true only within scope. The original gate was per-delta. The current gate is per-delta and accumulated-drift. The discipline still has uncovered scopes (for example, cross-repo cumulative state spanning tinkerclaw plus the brain repo), but the most consequential within-fork scope is now covered.

The honest framing: the discipline catches what it scopes. The contribution is naming scoping itself as a primitive — what the discipline interrogates matters as much as what it asserts.

There is a second axis of honesty the maturity claim must respect, and it is sharpened by the live open-source ecosystem. Every gate the recipe layer carries (§9) is *structural*: the `subagents-and-recipes.md` verify blocks assert that exports still exist, that the loader still wires turn-start auto-seed, that the kit files still parse as YAML carrying slug/title/-summary. None of it measures whether a recipe, *when it fires*, produces a better outcome than the one-shot prompt it replaces — which is the abstraction's entire reason to exist. So "the recipe library enforces itself" today means *enforces its structure*, not *enforces its value*. Two adjacent systems make this gap concrete by closing it on the value side: **marketingskills** (Corey Haines, `coreyhaines31/marketingskills`, ~33k stars) ships per-skill `evals/evals.json` — realistic prompts paired with assertion lists, runnable under an LLM judge — and **headroom** (Tejas Chopra, `chopratejas/headroom`, ~24.7k stars) ships a reproducible eval suite (`python -m headroom.evals`) that reports a measured accuracy delta (≈0 on GSM8K/TruthfulQA/SQuAD/BFCL) against its compression claims. The honest position: our structural-verify discipline is *stronger* than either system's — neither has a `verify:`-gated governing doc or commit-anchored invariants — but both carry a behavioral-eval surface we do not, and that surface is exactly what would let the "abstractions accumulate" thesis (§9.3) claim *measured retention-of-quality* rather than only measured structural integrity. A per-recipe assertion set, runnable as a judge, is therefore queued as the highest-value next gate, not folded into the present maturity claim. It belongs to the same priority logic as Constraint P: a claim of value, like a claim of privacy, must be backed by a gate that runs, not by prose.

8. The privacy addendum, summarised

One incident. One principle. One gate-hardening. The fix is tactical; the lesson is structural.

Per-delta enforcement is necessary but not sufficient. Any contract over absolute state needs an absolute-state check. The discipline was implicitly per-delta everywhere; this account names the shortcut and patches the highest-cost instance (privacy). The others (negative evidence, bug patterns) are known-shortcut and queued.

Priority axes are not co-equal. Privacy dominates functionality because irreversible-vs-reversible consequence demands asymmetric priority. Codified as design principle #17.

The gate is the last line of defence, not the first. The first line is the writer being aware of the rule. The discipline keeps both — principle in the bible, rule in two agents' memory, gate in the hook. Each necessary; none sufficient.

9. The recipe as intermediate abstraction

The intermediate-abstraction argument has so far been about *documentation*: a self-documenting codebase needs a missing middle layer between source code and a single prose mental model, and that layer is an atlas of `.md` files — one per question class — whose claims are enforced by executable `verify:` blocks. This section makes the same argument about *behavior*. The fork runs an orchestration substrate — recipes (on disk, kits) — and a recipe is the missing middle layer between a one-shot prompt and a script.

9.1 The layer the thesis predicts

The opening claim is that AI-pair coding operates with two levels and is missing the middle. For *behavior*, the two levels are:

- **The one-shot prompt — a wish.** You describe what you want once; the model improvises a path. Cheap to write, impossible to replay identically, no contract about what it does.
- **The script — brittle code.** A `.mjs` or shell pipeline that does exactly one thing the same way every time. Expensive to write against a moving runtime; goes stale the moment an RPC name changes.

A recipe sits between them: a named, replayable workflow expressed in the model's own idiom rather than in code, version-controlled, with explicit steps the model executes but does not re-derive. This is the function/module relationship the thesis claims is missing — the wish gets a name and a contract without being frozen into brittle code. A recipe is to a prompt what a function is to inline code: a unit between a single improvised utterance and a hard-coded procedure, durable enough to replay but loose enough to survive a moving runtime.

Canonical terminology: the conceptual unit is a *recipe*; the on-disk artifact in the fork is a `kit.md` file. The schema header is mid-migration — most files carry `schema: "kit/1.0"` and a growing set carry `schema: "recipe/1.0"`, both read by the same loader (a dual-read parser accepts either). This paper uses "recipe" for the abstraction and "kit" for the file, matching the dominant on-disk name.

9.2 The schema is a cross-registry interchange format, not a fork-local convention

The function/module analogy is only complete if modules cross project boundaries — a standard library is a real standard library precisely because its units travel between codebases that never coordinated. The recipe layer clears that bar, and not hypothetically. The on-disk `schema: "kit/1.0"` header this paper's loader reads is the *same* header published by an independent registry: the **Journey** kit registry (journeykits.ai) ships kits that are kit/1.0-native, and we ran a license-gate + attribute import pipeline that distilled four external Journey kits into our catalog. Both ends are present in the live tree: our `schema: "kit/1.0"` files under `extensions/tinkerclaw-prefrontal/kits/<slug>/kit.md`, and an installed Journey skill under `~/.agents/skills/journey/`. A behavioral intermediate abstraction authored in

one agent's registry parses and runs in another's because they share the schema contract — interop is therefore real and proven, not asserted.

This upgrades the central thesis. The claim is no longer "the fork ships a missing-middle behavioral layer" but "the missing-middle behavioral layer has a portable interchange format with at least two independent registries and a working import pipeline." That is the precise sense in which a recipe is a *stdlib unit* rather than a private macro.

The differentiation must stay honest. Journey is a *registry/marketplace* — its contribution is distribution and discovery, the catalog and the install path. It does not ship the enforcement spine this paper is about: the five primitives plus Constraint P, the `verify:`-gated governing doc (`subagents-and-recipes.md`), and the depth/cycle-guarded composition algebra (§9.3). Read together, the two systems divide the labor of a real stdlib cleanly: Journey (and the shared schema) demonstrate that the abstraction *travels*; we contribute the layer that makes it *enforce*. The import pipeline is itself gated — the license-gate + attribute step is a hard precondition before an external kit enters our catalog, the same shape as Constraint P applied to provenance rather than to PII.

A consequence worth stating: the fork's catalog is uncapped by design. Matching is code-side and only the winning recipe's plan is injected per turn (§9.4), so the marginal cost of carrying a kit is near-zero until it matches. This is *because* the schema is shared and cheap to carry, not in spite of it — a portable, low-cost-to-hold unit is exactly what makes an uncapped library tractable. The concrete budget shape behind this is visible in the marketingskills router discussed in §9.6: a ~130-token router replaces ~915 tokens of always-loaded skill descriptions while keeping all 44 frameworks reachable on demand — code-side matching turns library size into a near-free axis.

9.3 Abstractions built from abstractions: the composition seam

A layer is only a real abstraction layer if abstractions can be built from abstractions. In the fork, three sibling step directives provide that, each operating at a different point in the run:

The first is `uses:` (invocation): a step whose body opens with `uses: <kit-slug>` runs another kit as a sub-step — recipes calling recipes. This is verified shipped:

- The runner parses a leading `uses: <slug>` directive per step via `parseUsesDirective` (the directive regex sits at `recipe-runner.ts:245`, inside the function exported at line 243; bare slugs normalize to `globalcaos/<slug>`).
- Recursion is depth-guarded (`MAX_USES_DEPTH = 3`, the constant for sub-kit recursion at `recipe-runner.ts:191`) and cycle-guarded (the call stack is seeded with the kit's own ref so a self-`uses:` is caught at depth 0 — a fix that came directly out of review). A depth breach short-circuits the step with a "composition depth limit reached" note; a ref already on the stack short-circuits with a "composition cycle" note naming the chain.
- Real kits use it: `capability-survey/kit.md` invokes `uses: multi-modal-sweep` (line 78) and `uses: subagent-driven-dev` (line 104).

This is the invocation half of a composition algebra: a recipe calling a recipe, bounded so the call graph can't run away. It is the programming-language analogue the design predicts — function-calls-function, with a stack and a cycle check.

The second is `composes:` (inline step-merge), and it ships too. A kit's frontmatter declares `composes: [slug, ...]`; when that kit matches a prompt, the planner expands the composed slugs and pulls their steps into the merged plan before the kit's own steps (`buildMergedPlan` in `recipe-matcher.ts:436`, the per-entry expansion at line 452). Composed steps lead, so a composite reads as "do sub-recipe A, then sub-recipe B, then my own steps," deduped by normalised title so a shared step is not run twice. This is the same loop-prevention as `uses:`: expansion is depth-guarded (`depth > 3`) and cycle-guarded (an `expanded` set, so a kit pulled in

once is not pulled again). Where `uses:` is a runtime hop into a sub-recipe mid-run, `composes:` is a plan-build-time merge that inlines boilerplate steps without a runtime call.

The third directive is `invoke skill:<id>`, and it calls a *typed stdlib primitive* inline. It is parsed by `parseInvokeSkillDirective()` (`recipe-runner.ts`) and recognised in its two-word `invoke skill:` form by both IO-scanners — `recipe-types.ts` `OTHER_DIRECTIVE_RE` and `recipe-runner.ts` `leadingDirectives` — so directive order within a step is free. This directive is what makes composition-from-LIBRARY (as opposed to composition-from-other-recipes) deterministic: `prefrontal.recipe.compose` (`recipe-rpcs.ts`) calls `fork.skill.search`, emits one `invoke skill:` step per hit in rank order, validates, and persists the result stamped `authoredBy:"jarvis-on-the-fly"` — never clobbering a curated kit. So the composition algebra now has three operations, not two: `uses:` is a runtime hop into another recipe, `composes:` is a plan-build-time merge of another recipe's steps, and `invoke skill:` is an inline call into a typed library primitive. All three are bounded so the call graph cannot run away.

9.4 Where new abstractions come from: on-the-fly authoring

A function library that can only be used, never grown in the moment a gap appears, accretes slowly. The fork closes that loop. The matcher seeds a plan from the prompt at turn start (`seedPlanFromPrompt`, `recipe-matcher.ts:557`); when nothing clears threshold, it logs a NO-MATCH with an explicit `recipe-gap` marker and offers authoring (`recipe-matcher.ts:638`). After a kit is authored on the fly, the in-memory index cache is dropped (`invalidateRecipeIndexCache`, `recipe-matcher.ts:169`) so the new abstraction is immediately discoverable.

A correction belongs here, and it must be split by surface. For *recipe-matching*, matching is lexical by default; a semantic fallback lane exists but is gated default-OFF — it runs only when an embedding seam is injected (`embed?: EmbedFn`), and lexical-only is the default path. The lane fires inside an `if (deps.embed)` guard at `recipe-matcher.ts:612`; without that seam the matcher is byte-identical to pure lexical scoring. So the right claim for recipe-matching is not "the match is now semantic" but "the match has a semantic fallback that, when enabled, recovers paraphrases the lexical pass missed."

The companion surface — *skill-search*, which feeds composition-from-library (§9.3) — is no longer in that caveat, and the distinction matters because it is exactly the place where a tempting overclaim ("our matching is semantic") would be half-true. `fork.skill.search` (`src/fork/skill-rpc.ts`) resolves a real in-process embedding function via `resolveSkillEmbedFn` (`src/memory/engram/skill-embed.ts` — the same path the consolidation cron uses, so the embedder is shared rather than duplicated) and does batched-embed plus cosine ranking, with keyword scoring only as a fallback. This is a live embed provider, not a frozen one: deposited skills ranked at cosine 0.71 against a paraphrased query, which a keyword pass would have missed. The honest two-line statement is therefore: recipe-matching is lexical-first with a semantic lane default-OFF; skill-search is semantically live by default. Conflating the two would either understate skill-search or overstate recipe-matching.

A second correction guards the same thesis from the other direction. The "the library grows organically" claim depends on an extractor that, for a window, was silently inert: `detectEpisodes` always emitted `keyDecisions:[]`, so `isSkillWorthy` never passed and the live skill-library never actually grew. The fix is in `buildEpisode` (`src/memory/engram/episode-detection.ts`), which now derives `keyDecisions` from the episode's tool-call trace — two or more tool-call steps marks a multi-step procedure as worthy, a lone one-shot call is declined — while `isSkillWorthy` itself is unchanged. The organic-capture claim is true *now*; it was latent-broken before, and the paper records that rather than papering over it.

This is the abstraction-creation path the "abstractions accumulate" thesis requires. The gap that exposes a missing recipe is the same conversation that creates it: the model hits a NO-MATCH, authors the kit, and the next matching request finds it. The library grows at the point of need, in the model's idiom, and is version-controlled like any other intermediate

abstraction. A behavioral abstraction layer stays current this way rather than rotting toward folklore — the same fate the design-principles file describes for undocumented rules.

9.5 The schema's negative space: anti-triggers and load-context contracts

The recipe/kit schema described so far — slug, title, summary, steps, `composes:`, `uses:`, `invoke skill:`, `version:` — is entirely a *positive* contract. It declares what the abstraction does, how it composes, how it is matched, and (now) what its typed IO is. It says nothing about the abstraction's *negative space*: when the recipe must **not** fire, and where it may **not** run. §9.4's matching story is purely about getting a recipe to fire (lexical-first, NO-MATCH → author); it carries no symmetric story for *suppressing* a fire that would be wrong.

A directly comparable system makes the omission visible. **addyosmani/agent-skills** (Addy Osmani, `addyosmani/agent-skills`, ~56.8k stars) is a "named, gated, replayable unit" library for Claude Code plugins — the same abstraction class as our kits — and its per-skill schema carries two sections ours lack. The first is a **"When NOT to use"** anti-trigger section: an explicit applicability boundary the matcher is meant to respect to avoid mis-firing. The second is a **"Loading Constraints"** section declaring *where* the abstraction may load — main orchestrator versus subagent — with a documented degraded fallback when the preferred context is unavailable. Both are structural contracts on negative space: one on *when* the unit must not fire, one on *where* it may not run.

This is exactly the kind of executable-invariant the title concept ("intermediate *abstractions*") says the executable layer deserves, and its absence is a real hole in the §9.6 claim that both layers are gated by the same primitives. A `verify:`-gradeable anti-trigger field and a load-context field would let the matcher *refuse* as deliberately as it *fires*, and would let the governing doc assert that refusal-coverage exists — the same way it asserts that turn-start auto-seed exists. The honest comparison runs both ways: our recipes already carry something addyosmani's do not (runtime composition with depth/cycle guards, and on-the-fly authoring at the point of need), but their schema encodes an applicability/load-context contract ours does not. The next revision's mechanism work, not yet shipped, is to add an anti-trigger axis and a load-context axis to the kit schema and a verify block that asserts their presence — promoting the negative-space contract from prose convention to executable invariant. Until that lands, §9.6's "both layers gated identically" claim is held to "both layers gated identically *on positive structure*; negative-space gating is queued."

9.6 Why the two layers belong together

The documentation atlas and the recipe library are the same shape under the same primitives:

- **Multi-optic atlas / one recipe per task class.** One documentation file answers one question class; one kit answers one task class. The fork carries 38 kits with `schema: "kit/1.0"` (counted live on 2 June 2026), one per task class (debug, revise-paper, security-audit, upstream-merge, . . .). The library is uncapped by design (§9.2): code-side matching means an unmatched kit costs near-nothing, so the file count is free to grow. The concrete budget payoff is the one marketingskills demonstrates directly — its ~130-token router skill replaces ~915 tokens of always-loaded framework descriptions while keeping all 44 frameworks reachable on demand. That is the measured version of our "inject only the winner" claim: code-side routing converts an O(library-size) prompt cost into an O(1) one.
- **Executable invariants.** The documentation atlas's claims have `verify:` blocks; the recipe library's structural claims do too — `subagents-and-recipes.md` asserts via verify that `recipe-matcher.ts` still exports `seedPlanFromPrompt` and still warns on the recipe-gap, that `index.ts` still wires the turn-start auto-seed, and that the kit library holds at least ten `kit.md` files each parsing as YAML carrying slug/title/summary. The behavioral

layer is gated as the documentation layer is — on *positive structure*. Two gating axes are explicitly named as not-yet-covered: the negative-space contract (§9.5) and the behavioral-value contract (§7). Both are queued; neither is claimed as present.

- **Probe symmetry.** Kit execution emits recipe-state: the `onRecipeState` sink in `recipe-runner.ts` (the option declared at line 84) forwards to `fork.prefrontal.setRecipe` (wired in `recipe-rpcs.ts`), so the running abstraction is inspectable, not opaque — the write surface (running a kit) has its paired read (the RECIPES panel data source).
- **Commit-and-timestamp anchors.** The documentation file that governs the recipe library, `subagents-and-recipes.md`, carries `last_verified` / `last_verified_commit`, same as every other documentation file (`last_verified: 2026-06-02`, commit `06f8647fdc`). The `kit.md` files themselves carry a `version:` field rather than a verified-against-commit anchor; the commit-and-timestamp guarantee for the recipe layer lives in the governing doc's verify block, not in each kit.

The executable layer now also carries *types*, which is what makes the "intermediate *abstractions*" framing literal rather than metaphorical on the behavioral side. The `Skill` record (`src/memory/storage/types.ts`) gains optional `inputSchema?` / `outputSchema?` (JsonSchema) plus a flat `lineage?` (composedFrom / composedSkills / composedRecipes / sourceQuery); the fields are additive, so untyped prose skills round-trip unchanged. At execution (`recipe-runner.ts executeOnce`) an `invoke skill:` step adopts the skill's `outputSchema` and reuses the typed-output validate → budget-bounded-redispatch → persist path, and `compileSteps` eagerly lifts that schema so `checkPortWiring` validates downstream `in:` ports. Output is strictly validated; input is, for now, surfaced as prompt-text guidance rather than gated by a structured-input contract — a hard structured-input gate is deferred to the structured-call model. So the typed-IO story is honest in both directions: outputs are gated, inputs are guided, and the gap is named rather than implied closed.

The single new claim this section adds: intermediate abstractions for AI-pair coding are not only the descriptive ones (sequence, lifecycle, topology) but the executable ones (recipes). The descriptive abstractions answer "how does the system behave?"; the executable abstractions answer "how do I make it behave this way again, reliably, without freezing it into a script?" Both are mid-level, both are version-controlled, both rot into folklore without an enforcement gate, and both are now gated on positive structure — with negative-space and behavioral-value gating named as the queued remainder.

9.7 Related work: the recipe/kit ecosystem

This paper's contribution is a *gated, composable* behavioral middle layer; the open-source ecosystem around the same abstraction class is fresh enough that positioning against it belongs in the paper, not in a footnote. Four systems sit closest:

- **Journey** (journeykits.ai) — a kit/1.0-native registry and marketplace. Shares our on-disk schema, so kits authored there parse and run in our loader (§9.2). Contributes distribution and discovery; does not ship enforcement, composition guards, or `verify:`-gated governance. We import from it through a license-gate + attribute pipeline.
- **addyosmani/agent-skills** (Addy Osmani, ~56.8k stars) — a named/gated/replayable unit library for Claude Code plugins, the closest schema sibling. Carries negative-space contracts ours lack ("When NOT to use" anti-triggers, "Loading Constraints"); lacks our runtime composition algebra and on-the-fly authoring (§9.5).
- **marketingskills** (Corey Haines, `coreyhaines31/marketingskills`, ~33k stars) — a 44-framework skill library fronted by a ~130-token router, with per-skill `evals/evals.json` assertion sets runnable under an LLM judge. Demonstrates both the budget payoff of code-side routing we claim (§9.6) and the behavioral-eval surface we lack (§7).

- **headroom** (Tejas Chopra, `chopratejas/headroom`, ~24.7k stars) — a reversible-compression system whose relevance here is methodological: it ships a reproducible eval suite (`python -m headroom.evals`) reporting a measured accuracy delta against its own claims. It is the model for the value-gate §7 names as the highest-priority next addition.

The consistent shape of the differentiation: we are *ahead* on enforcement (no other system here has a `verify:`-gated governing doc, commit-anchored invariants, or a depth/cycle-guarded composition algebra) and *behind* on two contracts these systems already ship (negative-space applicability, and runnable behavioral evals). Naming both directions is the point — the maturity claim earns its credibility from where it concedes, not only from where it leads.

10. Conclusion

The structural lesson stands: per-delta enforcement is necessary but not sufficient, and privacy dominates functionality.

The thesis-level claim: the missing middle layer is not only the documentation atlas. It is also the recipe — the named, replayable, gated unit between a wish and a script. The fork ships that layer (kits), composes within it three ways — by invocation (`uses:`), by inline step-merge (`composes:`), and by inline typed-library call (`invoke skill:`), each depth- and cycle-guarded — and grows it at the point of need (on-the-fly authoring on a NO-MATCH recipe-gap). The same five primitives that make a self-documenting codebase enforce itself make the recipe library enforce itself.

The claim is now also *interoperable*: the recipe's `kit/1.0` schema is a cross-registry interchange format, shared with the Journey registry and exercised through a gated import pipeline, which completes the function/module-library analogy across project boundaries — a stdlib unit travels, and ours additionally enforces.

Two claims are held to the code. Recipe-matching is lexical by default with a semantic fallback gated default-OFF, not semantic by default; the companion skill-search surface, by contrast, is semantically live by default on a real in-process embedder — the two surfaces are distinguished rather than conflated. And the composition algebra is whole at three operations: a runtime hop (`uses:`), a plan-build-time merge (`composes:`), and an inline typed-library call (`invoke skill:`), all bounded so the call graph cannot run away.

The maturity claim is held equally to what it does *not* yet cover. Every recipe-layer gate today is structural; the negative-space contract addyosmani/agent-skills carries (anti-triggers, load constraints) and the behavioral-eval contract marketingskills and headroom carry (runnable assertion sets, measured accuracy delta) are both real gaps, named here as queued future work rather than absorbed into the present claim. The discipline is mature enough to be boring — and boring, here, includes being explicit about which gates have not been built yet.

The boring part now extends from how the system is described to how it is driven — and from a fork-local convention to a portable, multi-registry interchange format.

Appendix A — The five primitives plus one constraint

1. Multi-optic atlas.
2. Executable invariants.
3. Probe symmetry.
4. Commit-and-timestamp anchors.
5. Manifest-injection round-trips.

6. **Constraint P:** Privacy dominates functionality. Verify commands enforcing privacy contracts run with priority over functional ones. The pre-push gate scans accumulated drift, not just per-delta. The bypass is emergency-only. The same ranking logic governs abstraction provenance (the license-gate on external-kit import) and is the model for the queued value-gate (§7).

Appendix B — Sanitization-first command

```
git -C ~/src/tinkerclaw log -p origin/main..HEAD -- \
  ':(exclude)scripts/pii-pre-push.sh' \
  ':(exclude)TINKER_UI_DESIGN_BIBLE/pii-boundary.md' \
  ':(exclude)CLAUDE.md' \
  | grep -P '^\+[^+].*(<owner-first-name>|<owner-surname>|<owner-city>|/home/<user>|<employe
```

Zero hits required. This is the *manual* pre-merge audit a maintainer runs by hand before advancing `main`; it walks history (`log -p`) so it surfaces any commit that ever carried a token, which is the right behaviour for a human deciding whether to dig further. The pre-push hook's automatic scope 2 uses net-diff semantics instead, so it does not block a push on a leak develop already cleaned up. Self-referential files (the pii-pre-push.sh script and the bible's pii-boundary.md) are excluded by `:(exclude)` because they intentionally contain the regex pattern. Educational examples in other bible files use `<FirstName>` / `<owner-e164>` placeholder syntax to illustrate the rule without tripping the grep. The literal token list is in the live script; placeholders are used here so this paper does not itself become a leak source — an instance of principle #17.

Appendix C — §9 evidence map (verified 2 June 2026; cross-registry, three-directive, and skill-search deltas verified against live HEAD 24237e0cd22, 2026-06-04)

Claims in §9 were checked against the live fork:

- `uses:` runtime sub-kit invocation, depth- and cycle-guarded — `extensions/tinkerclaw-prefrontal/rec` (`MAX_USES_DEPTH = 3` at line 191; `parseUsesDirective` exported at line 243 with the directive regex at line 245; the call stack seeded with the kit's own ref for the self-`uses:` cycle guard). Real usage: `kits/capability-survey/kit.md` (`uses: multi-modal-sweep` line 78, `uses: subagent-driven-dev` line 104).
- `composes:` inline step-merge — shipped at plan-build time, not in the runner: `buildMergedPlan` in `recipe-matcher.ts` (function at line 436; per-entry composed-slug expansion at line 452) reads a kit's `composes: [slug, ...]` frontmatter and pulls the composed kits' steps into the merged plan ahead of the kit's own, deduped by normalised title, depth-guarded (`depth > 3`) and cycle-guarded (the `expanded` set). Transitive `composes:`/`uses:` deps are resolved during install (`recipe-rpcs.ts:539, 685`).
- `invoke skill:` inline typed-library call — `parseInvokeSkillDirective()` (`recipe-runner.ts`), recognised as the two-word `invoke skill:` form by both IO-scanners (`recipe-types.ts` `OTHER_DIRECTIVE_RE`; `recipe-runner.ts` `leadingDirectives`), so directive order is free. Compose-from-library: `prefrontal.recipe.compose` (`recipe-rpcs.ts`) → `fork.skill.search` → one `invoke skill:` step per hit in rank order → validate → persist `authoredBy:"jarvis-on-the-fly"` Typed IO at execution: `recipe-runner.ts` `executeOnce` adopts the skill's `outputSchema`;

`compileSteps` lifts it so `checkPortWiring` validates downstream `in:` ports. Output strictly validated; input surfaced as prompt-text guidance (structured-input gate deferred).
- Cross-registry schema (`kit/1.0`) — our `schema: "kit/1.0"` files under `extensions/tinkerclaw-prefro` share the header published by the Journey registry (journeykits.ai); an installed Journey skill is present at `~/.agents/skills/journey/`; four external Journey kits were distilled into the catalog through a license-gate + attribute import pipeline.
- On-the-fly authoring — `recipe-matcher.ts` `seedPlanFromPrompt` (line 557); NO-MATCH `recipe-gap` WARN plus authoring offer (line 638); cache drop after authoring via `invalidateRecipeIndexCache` (line 169). The extractor feeding organic growth: `buildEpisode` (`src/memory/engram/episode-detection.ts`) now derives `keyDecisions` from the tool-call trace (≥2 tool-call steps = worthy), fixing a prior window in which `detectEpisodes` emitted `keyDecisions:[]` and the library never grew; `isSkillWorthy` unchanged.
- Recipe-matching is lexical by default; the semantic fallback is gated default-OFF via an injected `embed?: EmbedFn` seam, firing only inside the `if (deps.embed)` guard at `recipe-matcher.ts:612` — held to "match has a semantic fallback when enabled," not "match is now semantic." Skill-search, by contrast, is semantically live by default: `fork.skill.search` (`src/fork/skill-rpc.ts`) resolves a real in-process embedder via `resolveSkillEmbedFn` (`src/memory/engram/skill-embed.ts`, shared with the consolidation cron) and does batched-embed + cosine ranking (live proof: deposited skills ranked at cosine 0.71), keyword only as fallback.
- Kit library: 38 kits with `schema: "kit/1.0"` counted live under `extensions/tinkerclaw-prefrontal/ki` every `kit.md` parses as YAML with slug/title/summary — `subagents-and-recipes.md` verify blocks (which assert ≥10 kits, not the exact 38).
- Recipe-state observability — `onRecipeState` sink in `recipe-runner.ts` (option at line 84) → `fork.prefrontal.setRecipe` wired in `recipe-rpcs.ts` (method call at line 917; asserted by `subagents-and-recipes.md` verify).
- Principle #17 string present — `design-principles.md` self-verify (verify block at line 16; heading at line 131), citing the 2026-05-13 incident.
- Two-scope PII gate — `scripts/pii-pre-push.sh`: scope 1 walks the push range commit-by-commit (`git log -p`), scope 2 scans the net diff `origin/main..${local_sha}` in diff mode (`scan_range ... "diff"`, scope-2 block begins at line 126), skipped when the pushed snapshot is already an ancestor of `origin/main` (the `merge-base --is-ancestor` guard at line 140). Diff mode is deliberate: it cancels add-then-remove so a cleaned-up historical leak does not false-positive.

Appendix D — Related-work delta (negative-space and behavioral-eval gaps, not yet gated)

The following contracts are carried by adjacent open-source systems and are **not** present in our recipe layer as of this revision. They are recorded here as the verified gap behind §7 and §9.5, so a later revision can mark each as shipped when its verify block lands:

- **Anti-trigger / "When NOT to use"** — per-skill applicability boundary in addyosmani/agent-skills; absent from our kit schema. Future mechanism: an anti-trigger field the matcher consults before firing, with a `verify:` asserting its presence.
- **Load-context / "Loading Constraints"** — per-skill main-vs-subagent load declaration with degraded fallback in addyosmani/agent-skills; absent from our kit schema. Future mechanism: a load-context field plus a verify that asserts coverage.
- **Per-recipe behavioral evals** — `evals/evals.json` assertion sets (marketingskills) and a reproducible eval suite reporting measured accuracy delta (headroom, `python -m`

`headroom.evals`); absent from our recipe layer, whose gates are all structural. Future mechanism: a per-recipe assertion set runnable under an LLM judge, gated as the value-counterpart of Constraint P (§7).

References