BROCA: Giving Agent Recipes a Grammar — A Gradual Type System and Combinator Algebra for Self-Composing Workflows

Oscar Serra, Jarvis · Independent Research

14 June 2026

Abstract

Broca's area is the brain's grammar engine. It is not where words are stored and not where meaning is felt; it is the structure that composes a finite vocabulary into an unbounded set of well-formed utterances, and that rejects the ill-formed ones before they are spoken. Damage it and the words survive but the *combination* collapses into telegraphic fragments. We take that as the design thesis of this paper: **a workflow language is only as powerful as its grammar — the rules by which small typed pieces compose into larger ones, checked before they run.**

An agent "recipe" — an ordered, portable workflow of steps an intelligent worker follows — is a good unit of operational knowledge, but in its plain form it is a sequence of English instructions, and English is a lossy wire format. A step's result reaches the next step as prose that the next step must re-parse, re-interpret, and hope it understood; nothing checks, before the run, that step seven actually consumes a field step three actually produces. This paper describes a small set of primitives that turn that lossy English-passing pipeline into a *typed, composable programming language* that an agent can author, type-check, compose, run, recover from failure inside, and sharpen against its own usage. The core is seven things. First, a **gradual typed value model with named ports**: a step may declare an `out:` schema and named `in:` ports bound from earlier steps' typed outputs, while any step that declares neither stays exactly the prose it was — typing is an overlay, never a rewrite. Second, **contracts at boundaries**: at plan-compile time, before any step executes, every input port and every guard reference is checked to resolve to a strictly-earlier producer whose output schema actually declares the referenced field. Third, a **boolean control grammar** — a `when:` guard evaluated by a tiny, sandboxed, side-effect-free expression evaluator (an OR-of-AND-of-comparisons over typed fields, explicitly *not* a JavaScript `eval`) and a `return:`/`done:` early-exit that closes the plan carrying the exiting step's typed value. Fourth, a **combinator algebra** — `map`/`filter` iteration over a typed array edge with a static or *dynamically-resolved* worker recipe, `compose` via the sub-recipe `uses:` edge, and `if-then-else` via the guard-plus-exit pair. Fifth, a **searchable skill standard library** an `invoke skill:` step resolves against, with the resolved skill's output schema adopted at compile so downstream ports type-check against the real contract. Sixth, **catchable recovery** — a classified-error taxonomy plus an `onError:` policy router (retry, fallback to another recipe, or continue-partial), with `done-partial` as a non-aborting terminal settlement. Seventh, **all bounds derived, never frozen** — re-dispatch, combinator fan-out, composition depth, and recovery-retry budgets are each computed from the live situation rather than read off a constant.

This is no longer a design that promises a target; it is a shipped language whose seven pillars are implemented and tested in the running substrate. We are precise throughout about what each pillar actually does, and about the two places where a mechanism is shipped but its strongest mode is held off by default (autonomous recipe self-editing is gated to *propose-only* unless an operator opts in) or scoped narrowly (the combinator algebra is the iteration combinators plus the guard/exit/compose primitives, not yet a closed set of four named higher-order recipes). We also situate the language against the live open-source ecosystem it now interoperates with — a context-compression layer (chopratejas/headroom) that answers the same lossy-prose problem from the opposite direction, an engineering-discipline plugin (addyosmani/agent-skills) and a marketing-skills corpus (coreyhaines31/marketingskills) whose authoring conventions sharpen ours, and a distribution registry (Journey / journeykits.ai) whose `kit/1.0` schema is the same grammar BROCA emits, making recipes a cross-registry interchange format rather than a private DSL.

Keywords: typed workflows, gradual typing, named ports, compile-time contracts, combinator algebra, sandboxed guard evaluation, early exit, skill library, catchable recovery, derived budgets, self-sharpening, self-composing agents, recipe-as-language, context compression, cross-registry interop

Contributions

1. **A gradual type system for prose workflows.** A per-step `out:`/`in:` directive vocabulary that adds named, schema-checked value edges to a recipe *without* converting it to code, where an undeclared step keeps its exact prose behavior — typing is opt-in per step and additive, not a migration.
2. **Compile-time contracts at boundaries.** A seed-time port-wiring check and a seed-time guard-reference check that each verify every consumer reference resolves to a real, strictly-earlier producer that declares the referenced field — turning a class of integration errors from deep-runtime failures into fast, precise authoring-time rejections.
3. **A combinator algebra over recipes.** `map`/`filter` iterate a worker recipe over a typed array edge (with the worker resolvable *dynamically* from a prior step's output), `compose` chains recipes through the depth-/cycle-guarded `uses:` edge, and `if-then-else` is the `when:`-guard / `return:`-exit pair — combinators are ordinary recipes, so they inherit fitness, archive, and versioning for free.
4. **A skill standard library as the composition vocabulary.** A searchable, versioned `SkillLibrary` (surface-embedding search with a keyword fallback, Laplace-smoothed fitness ranking) that an `invoke skill:` step resolves against; the resolved skill's output schema is adopted *at compile* so downstream `in:` ports type-check against the real contract, and a `recipe.compose` RPC mechanically assembles a recipe from the top-ranked skill hits.
5. **Catchable recovery with a classified-error taxonomy.** Every step failure is classified into one of a closed set of error kinds with a computed `recoverable` flag; an `onError:` policy router does retry (honored only for recoverable errors), fallback to a recovery recipe, or continue-partial, the last settling as a non-aborting `done-partial` that keeps the survivors.
6. **A self-sharpening loop over real usage.** A nightly consolidation lane reads per-step struggle from the live plan archive and proposes `rewrite_step_text` mutations for struggling steps; an apply path guarded by five rails (strict promote gate, authorship guard, snapshot-before-write, validate-or-skip, kill-switch) can enact them — but is *propose-only by default*, enacting nothing unless an operator sets the kill-switch.
7. **All bounds derived, not frozen.** Re-dispatch, combinator fan-out, composition depth, and recovery-retry budgets are each a function of live signals — value-of-work, historical confidence, prior effort spent, and affordability — never a hardcoded constant.

1. Introduction — From a Sequence of Wishes to a Grammar

1.1 The Broca analogy, taken seriously

A person with intact vocabulary but a damaged Broca's area can still name objects and still understand much of what is said to them. What they lose is *syntax*: the ability to bind words into grammatical structure. Speech becomes agrammatic and telegraphic — content words survive, but the scaffolding that joins them, and that signals who-did-what-to-whom, falls away. The lesson is that fluent composition is not the same faculty as a rich vocabulary. A system can have all the right pieces and still be unable to combine them reliably, because combination is its own competence, governed by rules that decide what is well-formed before it is ever uttered.

A language agent is, in exactly this sense, an extraordinary vocabulary with an impoverished grammar for *its own behavioral programs*. It can perform any single step brilliantly. But when steps are chained into a workflow, the joining is done in prose: step three writes "I found 4 failing tests in the auth module," step four reads that sentence and tries to extract the number 4 and the module name by re-reading natural language. There is no structure that *guarantees* step

four receives a field named `failingCount` of type integer; there is only the hope that the prose was clear and the re-read was faithful. And there is nothing that checks, before the workflow starts, that step four even refers to something step three produces. The agent has the words. It lacks the grammar.

This paper gives recipes that grammar. The thesis is that the right shape for a behavioral programming language is not "compile the recipe to code" (which sacrifices the homoiconic, agent-authorable, agent-mutable quality that makes recipes valuable) and not "leave the recipe as prose" (which sacrifices every guarantee). It is to add a *small typed combinatorial layer over the prose* — a value model, boundary contracts, a control grammar, a combinator algebra, a shared skill vocabulary, catchable recovery, and situational budgets — so the pieces compose with checked structure while the steps themselves stay readable, writable, and rewritable by the agent.

1.2 What a recipe is, and the gap this closes

A recipe is a structured workflow expressed as a portable markdown document: ordered steps, each a short instruction with optional tool hints and a success criterion, plus metadata about what it is for. A companion line of work establishes *why* this is the right unit — it is the named, replayable middle layer between a one-shot prompt (a wish that shatters on anything unforeseen) and a brittle script (code that executes mechanically and breaks on the first input it did not anticipate) — and ships a composition algebra in which one recipe can merge another's steps at build time or invoke a sub-recipe at runtime, each cycle- and depth-guarded. A second line of work establishes that this substrate *is* an agent's executive function: on every turn it matches intent to a recipe, composes from existing recipes, authors a new one when nothing fits, runs it with bounded loops, and regulates effort to the task. We take both as given and do not re-derive them.

What neither provides is a *type discipline* on the values that flow between steps, a *grammar* for composing steps conditionally, a *catalogue* of reusable typed primitives to compose from, a *recovery model* when a step fails, or a *feedback loop* by which the workflows sharpen against their own usage. That is the gap this paper closes. We are not building a new substrate; we are giving the existing one a type system, a combinator grammar, a skill vocabulary, a recovery semantics, and a fitness-driven self-sharpening loop.

1.3 What this paper claims, and what it does not

This paper claims the full typed-language stack: the value model and ports; the compile-time contracts; the combinator algebra; the skill standard library; catchable recovery; the self-sharpening loop; and the derived-not-frozen budgets that govern all of the above. Each is implemented in the running substrate and exercised by tests, and each is grounded below to a named mechanism.

Two honesty boundaries hold throughout. First, the self-sharpening loop's *autonomous apply* — the path that mutates a recipe's step text without a human in the loop — is shipped behind a kill-switch and is **off by default**; by default the loop *proposes* rewrites and applies nothing. We treat propose-only as the shipped behavior and the autonomous mode as an opt-in. Second, the combinator algebra as shipped is the iteration combinators (`map`/`filter` over a typed array edge, with dynamic worker resolution), `compose` (the `uses:` sub-recipe edge), and `if-then-else` (the guard/exit pair); it is not yet a closed library of four named, first-class higher-order recipes with a settled "I take recipe X as my then-branch" port shape. The mechanism is shipped; the last naming/packaging decision is the one open micro-design item, and we say so where it lives.

A third honesty boundary is new and methodological. We argue throughout that a typed value edge is cheaper and safer than a re-parsed prose carry-forward, but as of this revision we have **no measured token-savings or accuracy-delta benchmark of our own** for that

claim. The adjacent open-source baseline that *does* meet that bar — chopratejas/headroom, with a reproducible GSM8K/TruthfulQA/SQuAD/BFCL eval suite (Section 16.1) — sets a methodological standard this paper has not yet met for the value-model claim. We state the typed-edge efficiency argument as an argued result, not a measured one, and name the eval that would settle it.

2. The Lossy-Prose Problem

2.1 English is a wire format, and a bad one

When step *N* finishes, its result must reach step *N+1*. In a plain recipe that transport is a prose note. Prose is a magnificent medium for an instruction to a human-level worker and a terrible medium for a *value passed between machine steps*, for three reasons.

First, it is **unparsed**. The downstream step receives a paragraph and must itself perform extraction — find the count, find the verdict, find the path — with no schema to validate against and no error if it guesses wrong. Every edge in the workflow re-implements ad-hoc parsing, and every such parse can silently misread.

Second, it is **uncontracted**. Nothing, anywhere, states that step *N* promises to produce a `verdict` field or that step *N+1* requires one. The dependency exists only in the author's head and in the wording of two prose blocks that can drift apart independently. A rename in one place is not caught by the other.

Third, it is **unbounded and lossy under truncation**. A note long enough to carry the real result risks being summarized to fit a display or carry-forward budget, at which point the very field the next step needed may be the part that was cut. Carry-forward digests in the running substrate are deliberately bounded to a few hundred characters; a typed value must not be subject to that bound, precisely because it is the load-bearing payload, not a human-facing summary.

2.2 Two answers to the lossy-prose problem: eliminate the prose, or compress-and-recover it

There are two structurally different ways to defeat the lossy-prose transport, and naming both makes BROCA's choice precise rather than reflexive.

The first answer — BROCA's — is to make the prose **unnecessary at the load-bearing edges**: replace the paragraph with a typed, named-port value, so the downstream step binds `steps.3.out.failingCount` directly and there is no re-parse, no truncation risk, and no schema-less guessing on those edges at all.

The second answer is to keep the prose but make it **smaller and recoverable**. This is the design of chopratejas/headroom (Choprateja S. et al., 24.7k, Apache-2.0), a reversible context-compression layer built on **Compress-Cache-Retrieve (CCR)**: the originals are cached and the model retrieves them on demand, with per-content-type compressors — statistical JSON-array crushing (SmartCrusher), tree-sitter AST-aware code compression, and log/diff/text handlers — reported at 60–95% token savings with near-zero accuracy delta on a reproducible benchmark suite. headroom never needs to know what `failingCount` *is*; it crushes and caches the opaque note and serves it back when asked.

These are complementary, not competing, and BROCA's gradual typing (Section 3.2) is exactly why: most steps in a living library stay prose by design, and for those untyped steps a CCR layer is the right tool — it directly attacks the same few-hundred-character carry-forward truncation that Section 2.1 names as the third failure mode. BROCA eliminates the prose at the typed edges; headroom compresses-and-recovers it everywhere else. Section 16 develops this positioning and concedes the benchmark BROCA still owes.

2.3 Why "just write better prose" is not the fix

The reflexive answer is to instruct each step to emit cleaner, more parseable prose. This fails for the same structural reason that "just write more careful code without a type checker" fails: it moves the guarantee from the machine to the author's discipline, and author discipline does not hold across a growing, self-authored, self-mutated library of workflows. The moment an agent authors recipes on the fly and a nightly loop rewrites their step text, "the prose is careful enough" is a property no one is maintaining. The fix is the same fix that type systems are: make the structure explicit, make the boundary a contract, and check the contract mechanically — at the cheapest possible time, which is before the workflow runs.

3. The Gradual Typed Value Model

3.1 Ports and per-step IO

The value model is carried by two optional leading directives in a step's body. A step may declare `out:` — a JSON-Schema object describing the structured value the step produces — and `in:` — an array of named ports, each a small record `{ name, from, schema? }` where `from` is a reference of the form `steps.<n>.out[.<path>]` into a strictly-earlier step's typed output. The types and helpers for this vocabulary live in `recipe-types.ts`: the `Port` interface (`name`, `from`, optional `schema`), the per-step `StepIo` interface (`out?` / `in?`), `parseStepIoDirectives` (which reads the leading `out:/in:` directive block as JSON-typed values), and pure navigation/parse helpers for dotted-path lookup and `steps.<n>.out.<path>` reference parsing. The module is kept filesystem-free and side-effect-free on purpose, so the value-flow logic is unit-testable in isolation from the runner, and so the wire format has a single owner that both the runner and the recipe author depend on.

When a step declares `out:`, its canonical output becomes the validated object — persisted as a *structured* artifact rather than a prose digest. Runtime validation is `validateTypedNote` (in `recipe-runner.ts`): it extracts the fenced JSON block from the subagent's reply and validates it against the compiled schema with Ajv, returning either the typed value or the precise validator error text. Downstream, a port whose `from` is `steps.3.out.failingCount` binds the integer field directly, and a template reference `{{steps.3.out.failingCount}}` in a step's task text is replaced with the resolved value before the step is dispatched. The binding is over named, typed fields, not a prose blob — the edge carries data, not a sentence to be re-read.

3.2 Gradual: typing is an overlay, never a delete

The single most important property of this value model is that it is **gradual**. A step that declares neither `out:` nor `in:` behaves exactly as it did before: prose in, prose out. There is no migration, no rewrite, no "type everything or nothing." Typing is opt-in per step and additive.

This is enforced precisely, not aspirationally, in the directive parser. `parseStepIoDirectives` scans only the *leading* directive block of a step body, skips sibling directives it does not own (`uses:/loop:/when:/return:/done:/map:/filter:/onError:/invoke skill:`) so directive order never matters, and applies a deliberate rule: a line that begins `out:` or `in:` but whose value is **not** JSON-shaped is treated as ordinary prose and ends the directive scan — it is *not* parsed as a typed directive, and it does *not* throw. So a previously-valid untyped recipe whose body happens to begin with a sentence like "out: of scope, skip this when the cache is warm" keeps working unchanged; the line stays prose. A malformed JSON value on a line that *is* clearly a typed directive (looks like JSON but does not parse) is the one case that surfaces an error — because that is a real authoring mistake, not legacy prose. The design principle is *overlay, not delete.*

This gradualness is what makes the type system adoptable inside a *living* library. Recipes authored before the value model existed do not break; an agent can add a typed edge to one step of one recipe without touching the rest; and the nightly evolution loop (Section 8) can rewrite prose steps freely without tripping a type checker that was never asked to apply to them. It is also why the typed edge and a context-compression layer (Section 2.2) are complementary: the steps that stay prose under gradual typing are exactly the steps a CCR compressor is for.

4. Contracts at Boundaries — Compile-Time Checking

4.1 Port-wiring check

The value of a type is that it can be checked before it is trusted, and the cheapest time to check is before the workflow runs at all. At seed time — after the recipe's steps are parsed but before any step is dispatched — the runner builds a reduced view of each step (title, `out:` schema, `in:` ports, `when:` guard) and runs `checkPortWiring` (in `recipe-runner.ts`). For every `in:` port of every step it verifies, in order: that `from` parses as a `steps.<n>.out` reference; that the referenced step exists; that the referenced step is *strictly earlier* than the consumer (a forward or self reference is rejected — values only flow downstream); that the producer step actually declares an `out:` schema; and that the producer's schema declares the first path segment of the referenced field as a property. Each failure is a precise, human-readable message naming the consumer step, the port, and the exact defect. The function returns the list of errors; an empty list means the wiring is sound. The accompanying compile test exercises all four rejection paths and the happy path.

If any wiring error is found, the run is refused before a single subagent is spawned. This is the "contracts at boundaries" pillar in running code: a mis-wired recipe fails fast, at authoring/seed time, with an exact diagnosis, instead of failing deep inside a live multi-step run where the symptom (a step receiving `undefined` where it expected a count) is far from the cause (a typo in a port's `from`).

4.2 Guard-reference check and a skill's contract resolved at compile

The same discipline extends to control flow. A `when:` guard (Section 6) refers to earlier steps' typed fields, and those references can be wrong in exactly the same ways a port's `from` can be wrong. The seed-time guard-reference check mirrors `checkPortWiring`: for every reference token in a step's guard, it verifies the reference parses, points to an existing and strictly-earlier step that declares an `out:` schema, and that schema declares the referenced field. As with port wiring, any guard-reference error refuses the run at seed time with a precise message.

A subtle but important interaction lives here. An `invoke skill:` step (Section 7) that declares no explicit `out:` adopts the resolved skill's `outputSchema` **at compile time**, so the downstream `in:` ports of later steps type-check against the skill's *real* output contract rather than against nothing. The `uses:` sub-recipe edge resolves lazily at dispatch, but a skill's output contract is known statically and is therefore folded in before `checkPortWiring` runs. The contract checker thus sees the true producer schema whether the producer is an inline step, an invoked skill, or a port-bound value.

Both checks are deliberately **existence-only**: they verify that the referenced field is declared by the producer's schema, not that the comparison's operand types are mutually compatible. This is a conscious altitude choice — it catches the overwhelmingly common class of errors (typos, renames, wrong step numbers, references to fields nobody produces) cheaply and without a full type-inference engine, and it leaves richer type-compatibility checking as a clearly-scoped future tightening rather than a half-built present one.

5. Derived, Not Frozen — The Budget Principle

A programming language's *own* resource and recovery policy is part of its semantics. BROCA takes the stance that every such policy should be a function of the live situation, not a constant frozen at authoring time. A sibling line of work states the governing principle plainly: fixed thresholds, fixed lists, and fixed limits are legacy artifacts, because each one freezes a judgment about a world that then moves, and the artifact keeps enforcing the stale judgment while the system drifts into obsolescence; the remedy is to *derive* each decision from the live situation at the moment of use. BROCA applies this to all four of its bounds, each a small pure function with its own module so the derivation is testable in isolation.

- **Re-dispatch budget** (`deriveRedispatchBudget`, `redispatch-budget.ts`). When a typed step's output fails to validate against its `out:` schema, the runner re-dispatches with a corrective prompt carrying the validator's error text. How many times is `1 + min(3, ⌊requiredFieldCount/2⌋)` worth-of-work, scaled by `(0.5 + uncertainty)` where `uncertainty = 1  fitnessSuccessRate`, then clamped by affordability (`⌊remainingTokenBudget / estStepTokens⌋`) and floored at 1. More structure at stake earns more attempts (saturating, so a huge schema cannot run away); a shakier recipe earns more attempts; a thin budget cuts them.
- **Combinator fan-out** (`deriveCombinatorFanOut`, `combinator-budget.ts`). The width of a `map`/`filter` step is `min(arrayLength, ⌊remainingDispatchBudget / estIterationCost⌋)` — exactly as wide as the data, narrowed only when the budget cannot pay for the full array. No frozen iteration cap.
- **Composition depth** (`deriveUsesDepthBudget`, `combinator-budget.ts`). The maximum `uses:`/combinator nesting depth is `max(USES_DEPTH_FLOOR, ⌊remainingDispatchBudget / estIterationCost⌋)`, with `USES_DEPTH_FLOOR = 3` as the floor when no budget signal is threaded. The runner computes `MAX_USES_DEPTH = deriveUsesDepthBudget({})` — a derivation that defaults to the floor, not a literal `const MAX_DEPTH = 3`.
- **Recovery-retry budget** (`deriveRecoveryRetryBudget`, `recovery-budget.ts`). How many times an `onError: retry` policy re-runs a failed step is `max(1, ambition  priorAttempts)` where `ambition = 1 + round(2·uncertainty)`, again clamped by affordability. A reliable recipe still gets the floor of 1; a wholly unreliable one gets up to ~3 fresh attempts; each prior attempt shaves one off (diminishing returns).

In every case an author-supplied number, where one is allowed at all, is a *downward* cap on the derived bound, never the bound itself: `bound = min(authorN, derived)`, and a non-numeric or template-unresolved author value fails closed to the derived value. The language tries harder to get a high-stakes, low-confidence value right, and tries less hard when the work is cheap, the recipe is proven, or the budget is thin — by construction, not by a constant someone forgot to update.

A floored constant, where it appears at all, is a **safety ceiling, never the working value** — the working value is always the derived one, and the constant exists only to bound a pathological derivation. That distinction is the precise content of the derive-not-freeze principle, and we are careful to keep every literal in this section on the ceiling side of it.

6. The Control Grammar — `when:` and `return:`/`done:`

6.1 when: — a sandboxed boolean grammar, not eval

Conditional execution is expressed by a leading `when:` directive carrying a boolean expression over earlier steps' typed fields, for example `when: steps.1.out.passed == true`. The directive is parsed by `parseWhenDirective` and evaluated by `evaluateWhen` in a dedicated module, `when-eval.ts`, and the single most important fact about that module is what it is *not*: it is **not** a call into a JavaScript interpreter. It is a tiny, explicit, hand-written grammar.

The grammar is an OR of ANDs of factors (`and` binds tighter than `or`). Each factor may be prefixed with `not`; a factor is either `<operand> <op> <operand>` or a bare `<operand>` tested for truthiness. An operand is either a `steps.<n>.out[.path]` reference (resolved against the map of prior steps' typed outputs via the same dotted-path/reference helpers the value model uses) or a JSON literal (`true`, `false`, `null`, a number, or a quoted string). The comparison operators are `==`, `!=`, `<`, `<=`, `>`, `>=`; equality is a structural deep-equal, and the ordering operators require two numbers or two strings and otherwise raise a typed error. There are no parentheses, no arithmetic, no function calls, and crucially no path to arbitrary code — an operand that is neither a recognized reference nor a valid JSON literal raises a `WhenEvalError` rather than being executed. The module is pure and filesystem-free, exactly like the value model, so the guard logic is unit-testable in isolation.

This is a deliberate, security-relevant design stance: a workflow document that an agent authors and a nightly loop mutates must never become an injection vector. By making `when:` a closed grammar over typed fields and JSON literals rather than an `eval` of markdown, the expressiveness is bounded to precisely what conditional control needs, and the attack surface of "code hidden in a recipe body" is closed by construction.

The by-construction closure is the grammar's own guarantee, and it is the load-bearing one. It is now also backed by an independent runtime floor: a sibling line of work (the learned-intuition reflex gate) enforces a pre-execution policy on the *tool calls* any step — recipe-authored or freely improvised — is allowed to make, denying disallowed actions before they run rather than observing them after. The two guarantees are layered, not redundant: the closed `when:` grammar means a recipe body cannot *carry* code, and the reflex gate means a step cannot *invoke* a denied action even if its prose tried to. We note the second layer here for completeness; its mechanism, calibration, and its known over-flagging behavior on benign destructive-looking commands are the subject of the companion AMYGDALA work and are out of scope for this paper.

6.2 A guarded-off step is a successful no-op

The runtime semantics of a false guard are chosen carefully. When a step's `when:` evaluates false, the step is **skipped and settled as done** — it is *not* spawned, and it is *not* an error. The runner evaluates the guard against the live map of prior steps' typed outputs (gathered from earlier steps only), persists a note recording that the step was skipped and why, and settles it `done`. This is the right semantics for an if-style branch: the path not taken is not a failure of the program. A guard that *cannot be evaluated* — for instance because a referenced output is somehow absent at runtime despite the seed-time check — is, by contrast, a recorded step error (`guard-eval-error`), never a silent pass.

6.3 return:/done: — early exit with a typed value

The second control primitive is early exit. A bare leading `return:` or `done:` directive (parsed by `parseEarlyExitDirective`) marks a step as the plan's exit point: once that step completes, the plan closes and no later step group is dispatched. `return:` and `done:` are two spellings of the same marker. The early-exit `done:` is distinct from a human-facing prose `Done when:` success criterion a step may also carry; only a bare `done:` with nothing after the colon is the exit marker, so prose beginning "done: ship it tomorrow" is correctly *not* treated as an exit.

The exit is not a failure — it closes the plan as **success**, and it carries a value. The runner threads `early-exit` as a terminal settlement state alongside `done` and `error`; when a step in a group settles `early-exit`, the group-loop checks for it *before* the error path, closes the plan as `done`, and surfaces the exiting step's validated typed output as a `returnValue` on the run result. Early exit is cheap precisely *because* steps now have typed values: a workflow can decide, on a structured predecessor result, that it is finished, and hand back a typed answer rather than running to the end and hoping the caller can find the conclusion in the final prose.

Together, `when:` and `return:`/`done:` are the minimal control surface a workflow language needs: a guard to skip, and an exit to stop early with a result — and, as Section 7.3 shows, they *are* the `if-then-else` combinator. Heavier looping orchestration is intentionally *not* pushed into the markdown grammar; bounded loops and parallel groups are handled by the runtime, keeping the in-document grammar small and analyzable.

7. The Combinator Algebra — the Composition Layer

A combinator algebra is the layer the language is built around: operators that take recipes and produce composed behavior. BROCA ships the algebra as iteration combinators (`map`/`filter`), `compose` (the sub-recipe edge), and `if-then-else` (the guard/exit pair). Combinators are not a separate machine — they reuse the typed value model, the sub-recipe recursion, and the derived budgets already in place, which is exactly why they are cheap.

7.1 `map` and `filter` over a typed array edge

A `map:` or `filter:` directive names a `steps.<n>.out[.path]` reference that must resolve to an array, and the step's `uses:` directive names the **worker** recipe applied to each element. The parsers `parseMapIterDirective` and `parseFilterIterDirective` (in `recipe-runner.ts`) read the leading directive and validate it is a real step reference. At dispatch, `executeOnce` resolves the array from prior outputs, derives the fan-out width via `deriveCombinatorFanOut({ arrayLength })` (Section 5 — as wide as the data, narrowed only by budget), and runs the worker once per element as a *sibling* sub-recipe: each element gets its own sub-session keyed by `...::map::<stepIndex>::<i>` (or `::filter::...`), with the element bound into the worker's parameters as `item` and its position as `index`. Crucially the composition depth is incremented **once for the whole map step**, not once per element — a 50-element map is one level of nesting, not fifty — so the depth guard (`MAX_USES_DEPTH`) measures genuine nesting, not fan-out width.

For `map`, the worker's `returnValue` per element is collected into the result array. For `filter`, the element is *kept* when the worker's `returnValue` is truthy (a predicate-recipe filter); a lighter-weight `keep:` predicate over `{{item}}`/`{{index}}` is also supported, text-substituted and then evaluated by the same `evaluateWhen` engine the `when:` guard uses — so the filter predicate is the same sandboxed, non-`eval` grammar, not arbitrary code. The result of a `map`/`filter` step is itself a typed array value, so it can feed a downstream port like any other typed output.

7.2 Dynamic worker resolution — a recipe chosen by the data

The worker recipe need not be static. When the `uses:` reference is itself a `{{steps.n.out.path}}` template, `executeOnce` resolves it against prior typed outputs (`resolveKitRefTemplate`) into a concrete recipe reference before the iteration runs; an unresolvable template is a classified `map-filter-resolution` error, not a silent skip. This is the higher-typed edge the algebra needs: the recipe applied across a collection can be *selected by an earlier step's structured decision.* A planning step can emit `{ "worker": "owner/parse-invoice" }` and the map step

applies whichever worker the plan chose — the runner is acting as a recipe-factory dispatched off a typed value.

7.3 `compose` and `if-then-else` from existing primitives

`compose` — chaining recipes so one's typed output feeds the next's typed input — is the existing depth-/cycle-guarded `uses:` sub-recipe edge: a step `uses:` another recipe, whose typed output is bound by a downstream `in:` port. `if-then-else` is the `when:`-guard plus `return:`/`done:`-exit pair from Section 6: a guarded step that runs (or is a no-op) on a typed predicate, with an early exit carrying the typed result of the taken branch. Neither needs new machinery; both are named patterns over the typed value model, the sub-recipe recursion, and the control grammar.

Because every combinator is an ordinary recipe (or an ordinary directive over one), it inherits for free everything the substrate gives recipes — fitness measurement (Section 9), the never-delete archive, versioning. A map or filter pattern is written once and the whole library reuses it.

7.4 The one open packaging decision

What is shipped is the *mechanism*: iteration with dynamic worker resolution, compose, and the guard/exit pair, all tested. What is not yet frozen is the *packaging* of `if-then-else`, `map`, `filter`, and `compose` as four named, first-class higher-order recipes with a single settled port shape for "I take recipe X as my then-branch" — as a typed input port whose schema is a recipe reference, or as a dedicated combinator directive block. The `Port` interface already reserves the optional `schema?` field for exactly this higher-typed port, and the dynamic-worker path proves the runner-as-recipe-factory edge works; what remains is choosing the surface syntax and naming the four operators, a decision deliberately left to the algebra's first heavy real use rather than frozen before it.

8. The Skill Standard Library

A combinator algebra needs a vocabulary to compose. BROCA ships a searchable, versioned **skill standard library** that an `invoke skill:` step resolves against, so a recipe step can call a curated primitive by id instead of re-deriving it in prose.

8.1 The library: search, read, rank, fitness

The `SkillLibrary` interface (`skill-library.ts`) exposes `put`, `read`, `search`, `rank`, `recordOutcome`, `deprecate`, and `list`, over versioned skill records (each carrying steps, an `inputSchema`, an `outputSchema`, and an optional reference implementation). `search` does one batched embedding of `[query, ...skillTexts]` and ranks by cosine similarity when an embed function is wired, falling back to Jaccard token-overlap relevance otherwise — so search degrades gracefully to keyword matching with no embedder. Ranking folds in a Laplace-smoothed success rate so proven skills surface first, and `recordOutcome` compounds that fitness with real use.

It is worth being precise about what the embedding *is* here, because a sibling line of work (the AMYGDALA danger-head experiments) established a sharp negative result about what frozen sentence embeddings can and cannot carry. A frozen embedding is a competent **surface-similarity** signal — two skill descriptions that talk about the same thing land near each other in the space, which is all `search` asks of it — but it is *not* a carrier of deeper semantic judgment: an offline experiment found that a supervised head trained on top of a frozen MiniLM embedding scored *below chance* at classifying harmfulness, i.e. the geometry does not encode "is this dangerous." We therefore claim only what the geometry supports: the skill search is a relevance ranker over lexical/topical similarity, with the Laplace-smoothed fitness term

carrying the *quality* judgment the embedding cannot. The Jaccard fallback is not a degraded approximation of a smarter signal; it is the same family of surface-overlap relevance, which is why graceful degradation to it costs the ranker so little.

8.2 `invoke skill:` — procedure injection with a typed contract

A leading `invoke skill:<id>` directive (parsed by `parseInvokeSkillDirective`) is a sibling of `uses:`. At dispatch the runner reads the skill via `skillLibrary.read(id)` and injects its procedure into the step's task — the skill's steps, its reference implementation if any, and its expected input shape as a visibility hint — so the worker executes the curated procedure rather than improvising. The skill's `outputSchema` becomes the step's effective output contract (a step-level `out:` still *narrows* it), and, as noted in Section 4.2, that contract is adopted **at compile** so downstream ports type-check against the skill's real output. A skill's terminal outcome routes back to `fork.skill.recordOutcome` so the library's fitness compounds with every invocation.

8.3 `recipe.compose` — a recipe assembled from the library

The integration goes one step further: the `prefrontal.recipe.compose` RPC (`recipe-rpcs.ts`) mechanically builds a recipe from the library. It searches the skill library for a query, emits one `invoke skill:` step per top-ranked hit in rank order, validates the assembled spec, and persists it as an authored recipe (stamped `authoredBy: jarvis-*` so the authorship guard of Section 9 admits it). This is the compose-from-library primitive made concrete — deterministic and testable against a stubbed search — turning "find the relevant primitives and wire them" from an act of prose improvisation into a mechanical, typed assembly.

8.4 The library is internal; the grammar is an interchange format

The `SkillLibrary` is BROCA's *internal* composition vocabulary — it indexes the primitives one runner composes from. It is not a cross-organization distribution surface, and it does not need to be, because the recipe grammar itself turns out to be the distribution unit. The Journey registry (journeykits.ai) publishes workflows as **`kit/1.0` kits — the same schema BROCA recipes are written in** — and we ran a real **license-gated, attribution-stamped import pipeline** that distilled four external Journey kits into the live library. The grammar a second producer emits is the grammar BROCA consumes, which is the strongest available evidence that recipes are a genuine *language* and not a private DSL.

The division of labor is honest and clean. Journey is a **distribution registry**: it ships, indexes, and makes discoverable kits across organizations — a cross-org surface BROCA's internal `SkillLibrary` deliberately does not provide. Journey does *not* provide BROCA's type discipline: an imported kit gains compile-time port-wiring and guard-reference checks (Section 4), derived budgets (Section 5), and catchable recovery (Section 9) only **after** it crosses into our runner. So the two systems are complementary along exactly the axis the paper cares about — Journey answers *where a recipe comes from and how it travels*; BROCA answers *what guarantees it acquires once it is here*. This is the through-line of the recipe-as-language argument made concrete: a typed inter-step edge, then a typed external signature, then a private/public value split, and now a cross-registry interchange format that an independent registry already speaks.

9. Catchable Recovery and the Self-Sharpening Loop

9.1 A classified-error taxonomy

Every step failure in BROCA is **classified**, never raw. `recipe-types.ts` defines a closed `ErrorKind` set — `schema-mismatch`, `spawn-failure`, `timeout`, `budget-exceeded`, `guard-eval-error`, `sub-kit-failure`, `map-filter-resolution`, `depth-limit`, `skill-not-found`, `recovery-exhausted`, `fallback-failed`, `execution-error` — wrapped in a `ClassifiedError { kind, message, recoverable, details? }`. The `recoverable` flag is computed *from the kind*: a hard limit (depth, budget) has zero expected value from a retry and is marked non-recoverable, so the recovery router stays situation-derived rather than hand-set at each call site. No failure is silent: every error is classified, persisted on the plan step, and emitted on the trail.

9.2 `onError:` — retry, fallback, continue-partial

A step may carry an `onError:` policy with three modes (`OnErrorPolicy`): `retry` (with a count), `fallback` (to another recipe by ref), or `continue-partial`. The runner's recovery branch (`recipe-runner.ts`) routes a failed step through its policy *before* aborting:

- **retry** is honored *only* when the error is recoverable; the author's count is a downward cap on `deriveRecoveryRetryBudget` (Section 5), and a non-numeric count fails closed to the derived bound. A non-recoverable error skips retry entirely — retry has zero expected value against a hard limit.
- **fallback** resolves a recovery recipe ref (static or `{{...}}`-templated) and dispatches it through the same depth-/cycle-guarded `uses:` edge as a sub-session, with the fallback pushed onto the `_usesChain` so a fallback cycle is caught by the same guard.
- **continue-partial** persists the partial artifact and its error envelope, then settles the step as a non-aborting `done-partial`. In a `map/filter` step this is what lets the iteration *drop a failed element and keep aggregating survivors*: the survivor array is the step's value, the dropped indices are recorded in the classified error, and the plan continues.

9.3 `done-partial` settlement

`done-partial` is a terminal settlement distinct from `done` and `error`. When the recovery driver collects settlements, a `done-partial` does not abort the plan: it emits a `partial-completion` trail event (carrying the step's classified error) and the plan continues to the next group, with the step already persisted as a `done` row carrying its error envelope. This is the recovery analogue of the early-exit semantics: a survivable failure is a recorded, non-fatal outcome, not a crash.

9.4 The self-sharpening loop — propose-only by default

The recovery taxonomy feeds a feedback loop that lets the language sharpen its own workflows against real usage. The signal is per-step *struggle*, read from the live plan archive: when a plan closes, the plan store writes each run to `archive/<date>/<session>-<runId>.md` with per-step status and the classified error, and `step-struggle.ts` reads those back, aggregating per-step failures across runs and flagging a step as struggling when its failures clear a *derived* minimum-runs threshold and its failure rate clears a derived rate threshold (again, no frozen constants).

`recipe-optimize.ts` orchestrates the pipeline: read a recipe's archived plans, compute the struggle report, propose one `rewrite_step_text` mutation per struggling step (`recipe-evolution.ts`), and — *only when the kill-switch `RECIPE_AUTOAPPLY_ENABLED` is set* — apply each via `recipe-apply.ts`. A proposal is marked `autoPromotable` only on a strong signal: a *recoverable* dominant error kind over a *well-evidenced* step (failures $\geq 2\times$ the derived min-runs). The apply path is bounded by five rails: (1) a **strict promote gate** — only `autoPromotable` proposals reach it; (2) an **authorship guard** — only Jarvis-authored recipes (`authoredBy: jarvis-*`) may be

mutated, so hand-curated recipes are untouchable; (3) **snapshot-before-write** — the prior content is archived (never-delete) before any write; (4) **validate-or-skip** — a rewrite that fails `validateRecipeSpec` is dropped; (5) the **kill-switch** itself.

The dispatch is wired into the nightly consolidation cron (`engram-consolidate.ts`) through the recipe-evolution lane, which calls `prefrontal.recipe.optimize`. The honest default matters and is tested: **with the kill-switch off, the loop proposes and applies nothing.** The struggling step is identified and a `rewrite_step_text` proposal is produced, but no mutation is enacted unless an operator opts in. The self-sharpening *mechanism* is shipped end-to-end; its *autonomous* mode is an opt-in, not the default.

10. The Empirical-Fitness Loop

The budgets of Section 5 consume a recipe's historical reliability (`fitnessSuccessRate`), and the search ranking of Section 8 boosts proven skills. Both rest on an empirical-fitness loop that is shipped end to end across `recipe-rpcs.ts` and `recipe-fitness.ts`.

Producer. When a recipe runs, an `onTag` callback stamps a `recipe:<owner/slug>` attribution marker into the run's engram event store (`stampRecipeAttribution`), wrapped so that attribution can never break the run. The marker rides in the episode's events so it survives to consolidation.

Consolidation. During sleep consolidation, `attributeRecipe` extracts the `recipe:` tag from an episode's events (returning null when none is present — no false attribution), and `updateRecipeFitness` folds the episode into the recipe's record: increment runs, increment successes when the episode completed, and maintain a Laplace-smoothed `successRate` plus running averages of latency, token cost, and difficulty.

Consumer. When a recipe is dispatched, `makeFitnessLookup(engramBaseDir)(kitRef)` reads the recipe's empirical `successRate` from disk (returning `undefined`, which the runner reads as the neutral 0.5 default, when there is no record), and that value is passed into `runRecipe` as `fitnessSuccessRate`. It then flows into `deriveRedispatchBudget` and `deriveRecoveryRetryBudget`, so a recipe that has historically struggled earns more correction and recovery attempts, and a proven one earns fewer. The same lookup is wired into the recipe matcher's signals so search and matching rank proven recipes higher.

The loop closes: run → attribute → consolidate → fitness → budgets and ranking → next run. A recipe's reliability is measured from its actual outcomes and fed back into how hard the language tries on its behalf — confidence is observed, not assumed. The empirical-fitness term is also where the embedding's limits (Section 8.1) are answered: the geometry ranks by surface relevance, and *this* loop carries the quality judgment the geometry cannot.

11. Machine-Authorability

A workflow language for an agent must be one the agent can *write*, not only one it can run — otherwise the language is for humans and the agent is merely its interpreter. The authoring side lives in `recipe-author.ts`: `buildRecipeMd` assembles a complete recipe markdown document from a structured `RecipeSpec`, emitting the typed directives in the exact leading-directive form the runner parses (`out:`/`in:` as single-line JSON, plus the control directives), and `validateRecipeSpec` checks a spec before it is written to disk. The validator enforces the shape the rest of the language depends on: a traversal-safe slug, required title/summary/tags, at least one step, `out:` schemas that are object-typed, and ports whose `from` is a well-formed `steps.<n>.out[.<path>]` reference pointing at strictly-earlier steps. A round-trip property holds: a directive the author emits is one the runner's parsers read back.

This closes the loop the language thesis requires. The same agent that runs typed, contracted, guarded, recoverable workflows can also *emit* them — and emit them already conforming to the contracts, because the author validates against the same rules the runner enforces. The `recipe.compose` RPC of Section 8 is this in its strongest form: the agent does not even hand-author the steps; it composes them mechanically from the skill library. The grammar is not only checkable; it is generable. And, as Section 8.4 establishes, the grammar it generates is the same `kit/1.0` schema an independent registry already speaks — so an agent-emitted recipe is, without translation, a publishable interchange artifact.

11.1 Authoring conventions imported from the ecosystem

A second, smaller debt to the open-source ecosystem is methodological. The `validateRecipeSpec` shape above guarantees a recipe is *well-formed*; it does not yet guarantee a recipe declares *when it should and should not be chosen*. Two adjacent corpora supply the missing convention. addyosmani/agent-skills (Addy Osmani, 56.8k, MIT) — an engineering-discipline plugin packaging skills, subagent personas, commands, and hooks — standardizes a per-skill **"When NOT to use"** section and a **"Loading Constraints"** declaration (where a skill is allowed to load: main session vs. sub-agent). coreyhaines31/marketingskills (Corey Haines, 33k, MIT) — a 44-skill, 51-CLI marketing corpus — standardizes a per-skill user-phrase trigger description and an `evals/evals.json` acceptance set. Both conventions are directly applicable to BROCA's matcher surface: the substrate's recipe matcher already supports an anti-trigger field (`antiTriggers:` / `whenNotToUse:`) that *subtracts* from a recipe's lexical match score on an exact look-alike phrase, which is the runtime counterpart of Osmani's "When NOT to use." Adopting these as part of the authoring standard — declared triggers and anti-triggers, an evals set, and a load-site constraint — is the concrete next tightening of `validateRecipeSpec`, and it imports a discipline two independent corpora already pay rather than reinventing it.

12. Implementation Status and Verification

The honesty discipline of this work demands a precise ledger of what is shipped, what is shipped-but-gated, and what is shipped-mechanism-but-unfrozen-packaging.

Shipped and exercised.

- **The gradual typed value model and named ports** — `recipe-types.ts` (`Port`, `StepIo`, `parseStepIoDirectives`) and `recipe-runner.ts` (`validateTypedNote`): `out:`/`in:` directives, overlay-not-delete parsing, typed-field binding and `{{steps.n.out.path}}` resolution, and persistence of a validated step output as a structured artifact.
- **The compile-time contracts** — `checkPortWiring` (and the guard-reference check) reject forward/self references, missing producers, producers with no `out:` schema, and undeclared fields, with the run refused before any dispatch; an invoked skill's `outputSchema` is adopted at compile so downstream ports check against it. The compile test exercises all rejection paths.
- **The combinator algebra** — `parseMapIterDirective`/`parseFilterIterDirective`/`parseKeepDirect` plus the `executeOnce` fan-out: array-edge iteration, dynamic worker resolution via `resolveKitRefTemplate`, single-increment composition depth, `keep:` predicates over the `when:` engine, and survivor aggregation under continue-partial.
- **The skill standard library** — `skill-library.ts` (`search`/`read`/`rank`/`recordOutcome`, surface-embedding search with Jaccard fallback, Laplace ranking), `invoke skill:` resolution at compile and dispatch, fitness loopback, and the `prefrontal.recipe.compose` RPC.

- **Catchable recovery** — the `ClassifiedError` taxonomy, the `onError:` router (retry honored only for recoverable errors, fallback via the `uses:` edge, continue-partial), and `done-partial` non-aborting settlement.
- **The empirical-fitness loop** — `stampRecipeAttribution` (producer), `attributeRecipe`/`updateRecipe` (consolidation), `makeFitnessLookup`/`fitnessSuccessRate` (consumer into the budgets and the matcher).
- **All four derived bounds** — `deriveRedispatchBudget`, `deriveCombinatorFanOut`, `deriveUsesDepthBudget` (floor 3), `deriveRecoveryRetryBudget`, each varying with its live inputs and clamped by affordability, wired into the runner and the orchestration schema.
- **The control grammar** — `parseWhenDirective`/`evaluateWhen`/`WhenEvalError` and `parseEarlyExitDirective`, with the guard-skip-as-done and `early-exit`-closes-as-`done` settlements.
- **Machine-authorability** — `buildRecipeMd`/`validateRecipeSpec`, emitting directives the runner parses back.
- **Cross-registry interop** — the `kit/1.0` schema is shared with the Journey registry; a license-gated, attribution-stamped import pipeline distilled four external kits into the live library, which then acquired the contracts, budgets, and recovery of the runner.

Shipped but off by default. The self-sharpening loop's *autonomous apply* is gated behind `RECIPE_AUTOAPPLY_ENABLED`. The struggle reader, the proposer, the five-rail apply path, and the cron lane are all implemented and the propose path is tested; the autonomous-mutation mode enacts nothing unless an operator turns it on. By default the loop proposes and applies nothing — that is the verified default, not an aspiration.

Shipped mechanism, unfrozen packaging. The combinator *mechanism* (iteration, dynamic worker, compose, guard/exit) is shipped; the *packaging* of `if-then-else`/`map`/`filter`/`compose` as four named first-class higher-order recipes with a single settled recipe-reference port shape is the one open micro-design decision (Section 7.4), deliberately left to first heavy use.

Not yet measured. The value-model efficiency claim — that a typed edge costs fewer tokens and loses less information than a re-parsed prose carry-forward — is argued (Sections 2, 16) but not benchmarked. A reproducible eval to the standard headroom sets (Section 16.1) is named future work, not a present result.

13. Limitations and Negative Results

We name the language's present boundaries plainly; each is a deliberate scope line or a genuine limitation, not an oversight.

1. **Compile-time checking is existence-only.** The contracts verify that a referenced field is *declared*, not that its type is *compatible* with how it is used in a comparison or bound to a downstream schema. A guard comparing a string field with `>` to a number is caught at runtime (the evaluator raises a typed error), not at seed time. Full type-compatibility checking is future work.

2. **The `when:` grammar is intentionally minimal — and brittle at one seam.** It has no parentheses, no arithmetic, and no function calls; complex conditions must be expressed as OR-of-AND or pushed into the runtime. More sharply: because the boolean keywords split on surrounding spaces, a string literal that itself contains the substrings `and` or `or` is mis-tokenized. This is a documented, accepted limitation of the current grammar, not a hidden bug — a future grammar should tokenize before splitting.

3. **The combinator algebra ships as mechanism, not as a closed named set.** `map`/`filter`/`compose`/`if-then-else` all work, but the surface syntax that packages them as four first-class higher-order recipes with a single recipe-reference port shape is unfrozen by design (Section 7.4). Today a combinator is assembled from the iteration directives plus `uses:`/`when:`/`return:`, not invoked as `compose(f, g)`.

4. **Autonomous recipe self-editing is off by default.** The self-sharpening loop (Section 9.4) proposes rewrites unconditionally but applies them only when an operator sets `RECIPE_AUTOAPPLY_ENABLED`. The shipped default is propose-only. This is a safety stance, not a missing feature — but it does mean the closed-loop "the library rewrites itself overnight" behavior is opt-in, and we do not claim it runs unattended out of the box.

5. **Filter's two predicate paths have different power.** The `keep:` predicate is the sandboxed `when:` grammar over `{{item}}`/`{{index}}` (cheap, no spawn); the predicate-recipe path runs a full worker per element (expressive, costly). An author chooses; there is no automatic promotion between them.

6. **Some fitness inputs are best-effort.** The empirical-fitness loop is wired end to end, but a recipe with no archived runs reads as the neutral 0.5 default, so on a recipe's very first runs the budgets vary only with value-of-work and structure, not with confidence. Confidence sharpens as runs accumulate; it is not available a priori.

7. **Early exit and parallel groups interact subtly.** If a step carrying `return:`/`done:` shares a parallel group with other steps, the early-exit wins and a sibling error in that same group is not surfaced. The mitigation is an authoring convention — put an early-exit step in its own sequential group — not an enforced rule.

8. **The value-model efficiency claim is unmeasured.** We argue a typed edge is cheaper and lossless versus a prose carry-forward, but we ship no token-savings/accuracy-delta benchmark of our own. The adjacent open-source baseline (headroom, Section 16.1) meets that bar with a reproducible suite; BROCA does not yet, and we mark the claim as argued, not measured.

9. **Skill search ranks surface similarity, not meaning.** The embedding under `SkillLibrary.search` (Section 8.1) is a relevance ranker over lexical/topical overlap, not a carrier of deeper semantic judgment — a sibling negative result found a supervised head over a frozen embedding scoring below chance on a harmfulness task, so we do not lean the embedding to carry quality. Quality is carried by the empirical-fitness term, not by the geometry.

14. A Worked Example

To make the grammar concrete, here is a recipe that reviews a batch of pull requests, classifies each, fixes the auto-fixable ones, and exits early when none need attention — using ports, a guard, an early exit, a `map` combinator with a dynamic worker, an invoked skill, and catchable recovery.

```
### 1. Gather open PRs
out: {"type":"object","properties":{"prs":{"type":"array"},"count":{"type":"integer"}},"required":["

List the open pull requests for the repo. Emit a JSON object with `prs` (array of
{number, title, diffUrl}) and `count`.

### 2. Stop if there is nothing to do
```

```
in: [{"name":"count","from":"steps.1.out.count"}]
when: steps.1.out.count == 0
return:

Nothing to review. Return the empty result.

### 3. Triage the batch and choose a worker
in: [{"name":"prs","from":"steps.1.out.prs"}]
out: {"type":"object","properties":{"worker":{"type":"string"},"items":{"type":"array"}},"required":
invoke skill: triage.classify-prs

Classify each PR. Emit `worker` (the recipe ref to apply per item, e.g.
"owner/autofix-lint") and `items` (the array to map over).

### 4. Fix each PR
map: steps.3.out.items
uses: {{steps.3.out.worker}}
onError: {"mode":"continue-partial"}

Apply the chosen worker to each item; drop any element that fails and keep the rest.
```

Step 2's `when:` guard and bare `return:` are the `if-then-else` early branch: on an empty batch the plan closes as success carrying the typed empty result, and step 3 onward never runs. Step 3 invokes a library skill whose `outputSchema` is adopted at compile, so step 4's `map:`/`uses:` references type-check against it at seed time. Step 4 is the `map` combinator with a *dynamic* worker — the recipe applied per element is whichever ref step 3's plan chose — and its `onError: continue-partial` makes a per-element failure a dropped survivor, not a plan abort. The fan-out width is derived from the array length clamped by budget; the composition depth ticks once for the whole map; and if step 1's typed output had failed to validate, the re-dispatch budget would have been derived from the schema's required-field count and the recipe's historical fitness. Every pillar of the language is doing visible work in four steps of markdown.

15. Related Work

We position BROCA against the live open-source ecosystem it now interoperates with, treating these systems as first-class peers rather than footnotes — they are fresher than any static citation and, in two cases, BROCA shares a schema or a methodological bar with them.

Context compression for agents — chopratejas/headroom (Choprateja S. et al., 24.7k, Apache-2.0). headroom attacks the same lossy-prose-transport problem from the opposite direction (Sections 2.2, 16): rather than eliminating the prose at typed edges, it keeps the prose and makes it smaller and recoverable via Compress-Cache-Retrieve. The two are complementary — BROCA's typed edge for the load-bearing fields, headroom's CCR for the steps gradual typing deliberately leaves as prose. headroom is also the methodological foil for this revision: its reproducible GSM8K/TruthfulQA/SQuAD/BFCL eval suite is the bar BROCA has not yet met for its own value-model efficiency claim (Section 16.1).

Engineering-discipline packaging — addyosmani/agent-skills (Addy Osmani, 56.8k, MIT). A single plugin bundling engineering skills, subagent personas, command definitions, and hooks. Its contribution to BROCA is conventional, not architectural: the per-skill **"When NOT to use"** and **"Loading Constraints"** sections (Section 11.1) are precisely the declared-anti-trigger and load-site discipline BROCA's matcher already supports as an anti-trigger field, and adopting them as part of `validateRecipeSpec` imports a convention rather than reinventing it.

A marketing-skills corpus — coreyhaines31/marketingskills (Corey Haines, 33k,

MIT). A 44-skill, 51-CLI corpus whose per-skill user-phrase triggers and `evals/evals.json` acceptance sets are the second source of the authoring-standard tightening in Section 11.1. Its integration into our substrate (a vendored clone behind a single router skill plus a BROCA `marketing` recipe category) is an instance of recipes-as-vocabulary at corpus scale.

A distribution registry — Journey / journeykits.ai. Journey publishes workflows as `kit/1.0` kits, the same schema BROCA recipes are written in (Section 8.4). It is the cross-org distribution surface BROCA's internal `SkillLibrary` is not, and it does not provide BROCA's type discipline — the two compose cleanly along the "where a recipe travels" vs. "what guarantees it acquires" axis. A real license-gated import pipeline (four kits) is the existence proof that the grammar is a portable interchange format.

The companion lines of work this paper builds on — the recipe-as-substrate argument, the agent-executive-function argument, the derive-not-freeze principle, and the learned-intuition reflex gate — are cited in-text at the sections where they are used (Sections 1.2, 5, 6.1) and are not re-derived here.

16. Positioning Against Context Compression — and a Benchmark We Owe

16.1 Eliminate the prose vs. compress-and-recover it

Sections 2.2 framed the two answers to the lossy-prose problem; here we make the comparison sharp enough to be a real positioning claim and to name what it costs us.

The two systems differ in *what they know about the payload*. headroom's CCR operates on an **opaque** blob — it never knows that the note contains a field called `failingCount`; it crushes the bytes with a content-type-appropriate compressor (statistical for JSON arrays, AST-aware for code), caches the original, and serves it back on demand. BROCA's typed edge operates on a **named, schema-checked** field — it knows `failingCount` is an integer the producer declared and the consumer binds, so at that edge there is no compression, no cache, and no retrieval round-trip, because there is no prose to begin with. Stated plainly: **headroom makes the prose smaller and recoverable; BROCA makes the prose unnecessary at the typed edges.** Neither subsumes the other. A living BROCA library is mostly prose by design (gradual typing, Section 3.2), and a CCR layer is the right tool for exactly those untyped steps and for the few-hundred-character carry-forward truncation Section 2.1 names as a first-class failure mode.

16.2 The benchmark BROCA owes

The honest cost of this comparison is methodological. headroom backs its compression claim with a reproducible eval suite — GSM8K, TruthfulQA, SQuAD, and BFCL, runnable as a single `python -m headroom.evals` command, reporting token savings against an accuracy delta. BROCA backs its value-model claim — that a typed edge is cheaper and lossless versus a prose carry-forward — with an *argument from structure*, not a measurement. That is a real gap. The next concrete step for the value model is a comparable, reproducible benchmark: a fixed multi-step workflow run two ways (prose carry-forward vs. typed `out:`/`in:` edges), measuring token cost at each edge and downstream task accuracy, on a task set broad enough to be credible. Until that exists, "the typed edge saves tokens" is an argued claim, and we mark it as such (Section 13, item 8). headroom's suite is the standard we are holding ourselves to.

17. Conclusion

Broca's area does not make us eloquent; it makes us *grammatical* — it is the small, strict competence that turns a vocabulary into a language by governing how pieces combine and rejecting the combinations that are not well-formed. An agent that chains steps in prose has the vocabulary and lacks that competence: its values cross step boundaries as sentences to be re-read, and nothing checks, before it runs, that the chain even fits together.

BROCA supplies the missing grammar, and it is built, not promised. A gradual typed value model lets a step declare what it produces and consume named, typed fields from its predecessors, while every untyped step keeps its exact prose behavior. Compile-time contracts check, before anything runs, that every input port and every guard reference resolves to a real, earlier producer that declares the field — and a skill's output contract is folded in at compile so even composed steps type-check. A `when:` guard backed by a closed, sandboxed boolean grammar (emphatically not an `eval`) and a `return:`/`done:` early-exit carrying a typed value give the minimal control surface — and *are* the if-then-else branch. A combinator algebra maps and filters a worker recipe across a typed array, with the worker selectable by the data itself, and composes recipes through a depth-guarded sub-recipe edge. A searchable skill standard library gives the algebra a vocabulary, and a `compose` RPC assembles recipes from it mechanically. Catchable recovery classifies every failure and routes it through retry, fallback, or continue-partial, so a survivable failure is a recorded `done-partial`, not a crash. A nightly loop reads per-step struggle from the live archive and proposes sharper step text — applying it autonomously only when an operator opts in. And every bound the language enforces — re-dispatch, fan-out, depth, recovery-retry — is derived from value-of-work, confidence, effort, and affordability, never frozen to a constant.

This revision also places BROCA in its live ecosystem. The lossy-prose problem has two complementary answers, and an independent, benchmarked context-compression layer (headroom) is the right tool for the prose BROCA deliberately leaves untyped — which also names the reproducible benchmark BROCA still owes for its own typed-edge efficiency claim. Authoring conventions from two large open-source corpora (agent-skills, marketingskills) sharpen the recipe-matcher standard rather than being reinvented. And, most consequentially for the central thesis, the recipe grammar is the same `kit/1.0` schema an independent distribution registry (Journey) already speaks: a second producer emits the language and a real license-gated pipeline imports it, which is the strongest available evidence that this is a genuine *language* and not a private DSL.

We have been precise about the lines that remain: autonomous self-editing is shipped but off by default; the combinators are shipped as mechanism with their first-class naming deliberately unfrozen; the value-model efficiency claim is argued but not yet measured; and the skill embedding ranks surface similarity, not meaning, with quality carried by the fitness loop rather than the geometry. Those lines are not hedges — they are the same discipline the type system itself embodies. A grammar earns trust by checking its claims before it acts on them. So should the paper that describes it.

References