STRAP: A Structured Task and Recovery Agent Protocol for Reliable AI Agent Harnesses

Gautam Parab

Abstract Most AI agent failures are not reasoning failures; they are missing infrastructure around a capable model. An agent that cannot state what it is trying to do, cannot tell what changed in its environment, cannot prove its work is finished, and cannot classify why a step failed will behave unreliably regardless of model quality. We call the system that supplies this missing infrastructure a harness, and we present STRAP (Structured Task and Recovery Agent Protocol), a small, dependency-free specification and reference skill suite for building one. STRAP defines eight required surfaces — Contract, Context, Tools, State, Policy, Verification, Recovery, and Observability — and operationalizes four of them as installable, model-agnostic skills: a Task Contract Generator that converts vague requests into a bounded, checkable specification before any action is taken; a State Compiler that converts a raw tool-call transcript into durable Facts/Decisions/Progress/Lessons records instead of relying on context-window replay; an Adversarial Verifier that evaluates a finished artifact against its contract with an explicit incentive to reject rather than to agree; and a Failure Router that classifies a tool or verification failure into one of a fixed set of classes and dispatches a matching recovery strategy instead of a blind retry. We give the protocol's data schemas, the skill interfaces, a worked failure-classification table, two small reproducible experiments on the classification and routing mechanisms, and a discussion of where a harness this size is sufficient versus where it is not. STRAP is released as a set of plain-text skill definitions with no required framework, model provider, or database, on the premise that a specification developers can read completely in one sitting is one they will actually adopt.

1. Introduction

An AI agent is not a model. A model can reason over a prompt and produce a response; an agent must additionally locate the relevant part of a real environment, choose and operate tools correctly, carry state across steps that no longer fit in a context window, respect limits on what it is allowed to do unattended, tell the difference between a plausible-sounding result and a verified one, and decide what to do when a step fails. None of that is a property of the model weights. It is a property of the system built around the model — what we, following recent practitioner usage, call the **harness**.

The failure mode this paper is about is specific and common: a team ships an agent, it fails on some fraction of runs, and the team's response is to edit the prompt. The prompt gets longer. The failure rate barely moves, because the actual defect was never in the prompt. The agent did not know which files were in scope; it had a tool it was not supposed to use in that situation; it lost a decision made three tool calls ago because the transcript scrolled past the model's attention; it declared success

without running anything that could have disproven it; or it hit an error and retried the identical action with the identical arguments. Each of these is an environment defect, and each has a narrow, mechanical fix that does not require a smarter model.

Harness engineering, as a discipline, treats these defects as a design surface rather than a training problem: a task should arrive as a contract, not a sentence; tools should be gated, not piled on; state should be compiled into durable records, not replayed from a transcript; completion should be argued from evidence, not asserted by the model that did the work; and recovery should be routed by failure class, not repeated blindly. This framing has circulated in practitioner writing on harness engineering, but it stops short of an artifact a team can actually install, and — as §2 details — its individual pieces are each backed by a distinct strand of published agent research rather than by a single unifying study. This paper's contribution is that artifact.

We present **STRAP** — Structured Task and Recovery Agent Protocol — a specification for four of the eight harness surfaces (Contract, State, Verification, and Recovery), plus a schema for the remaining four (Context, Tools, Policy, Observability), together with a reference implementation as four Claude Code / LLM-agent skills. Each skill is a plain-text instruction file with a fixed input/output contract; none require a specific model provider, a vector database, an orchestration framework, or a hosted service. The design goal is a harness a team can have running, if minimally, before the end of the day it was proposed.

We make four contributions:

1. **A minimal, named protocol** (§3) for the four harness surfaces most responsible for the failure modes above, specified as literal data schemas — including a stable per-item identifier scheme that survives rewording — rather than prose principles. We ship one reference implementation (§4) plus a second, independently-authored one (`second-implementation/`, no shared code with §4) that parses a real contract, runs real acceptance checks as subprocesses, and produces schema-conformant Verdicts and routed recovery actions against a real fixture — schema-level interoperability, confirmed by a runnable conformance suite, not merely designed for. That second implementation's independently-designed classifier (a different algorithm from §6.1′s reference classifier, not a copy) agrees with the ground-truth labels on 39 of the same 48 corpus cases (81.2%), against the reference classifier's disclosed 88.0% on 50 cases (§6.1) — real, unedited evidence that two honestly-different implementations of the same seven-class contract land in the same neighborhood, not evidence they agree in general; see `second-implementation/README.md` for exactly what is and is not demonstrated by this comparison.
2. **A reference skill suite** (§4) — `task-contract`, `state-compiler`, `adversarial-verifier`, `failure-router` — implementing the protocol as installable agent skills, with a worked example in each skill file.
3. **Two small, reproducible experiments** (§6) isolating the failure-classification and recovery-routing mechanisms with disclosed confidence intervals and an explicit corpus-construction threat, rather than a single conflated "it works" claim.
4. **An explicit scoping discussion** (§5, §7) of when this four-surface subset is sufficient and when a team needs the remaining four surfaces (Context, Tools, Policy, Observability) or a heavier

orchestration layer, so that adopting STRAP does not become its own instance of the over-engineering it is meant to prevent.

2. Related Work and Terminology

The premise that agent reliability is a systems problem rather than a purely model-capability problem has precedent in production ML: model-serving infrastructure, feature stores, and evaluation harnesses in classical MLOps exist for exactly this reason — a good model in a bad pipeline ships bad predictions. Agent harnesses generalize this to the agentic setting, where the "pipeline" now includes tool selection, multi-step state, and unbounded natural-language task specification. Four independent lines of empirical agent research motivate three of STRAP's four implemented surfaces and its Policy surface; the Recovery surface's routing rule is instead evaluated directly in §6.

Tools and action grounding (Surface: Tools/Contract). ReAct [1] showed that interleaving explicit reasoning traces with tool calls, rather than reasoning alone, reduces hallucinated actions and compounding errors in interactive environments — early evidence that an agent's failure mode is often the *action*, not the reasoning that preceded it, which is the premise behind gating actions on a validated contract before they are taken. Toolformer [2] demonstrated that a model can learn, via self-supervised bootstrapping filtered by loss reduction, which API calls to insert into its own generations and with what arguments — competence for producing a call, learned inside the model, with no external gate on whether an individual call is valid before it executes. STRAP's Contract and Tool surfaces put that gate outside the model instead, in the harness, on the premise that a competent model can still occasionally produce an invalid or out-of-scope call and a harness should catch it deterministically rather than rely on the model's own learned judgment every time.

Self-correction and verification (Surface: Verification). Self-Refine [3] and Reflexion [4] both showed that a model critiquing and revising its own output over several rounds can improve task performance, using the same model and largely the same context for both generation and critique. Huang et al. [5] subsequently found that this same setup — an LLM correcting its own reasoning without external, ground-truth feedback — does not reliably improve and can degrade correct answers, isolating *intrinsic* self-correction (same model, same context, no external signal) as the failure mode. Zheng et al. [6] separately documented systematic biases in LLM-as-judge evaluation, including a preference for its own generation style. Together these three results are the direct empirical motivation for STRAP's Adversarial Verifier taking only the contract and the artifact as input and explicitly excluding the worker's own reasoning trace: the literature shows self-critique without an external check or evidence source is the specific configuration that fails.

Durable memory (Surface: State). MemGPT [7] and the generative-agents architecture [8] both replace transcript replay with an explicit, external memory store the model reads and writes to across a context-window boundary, and both report that this materially extends the length and coherence of tasks and simulations an LLM agent can sustain relative to keeping everything in the prompt. STRAP's four-bucket Facts/Decisions/Progress/Lessons schema is a narrower, more structured instance of the same architectural move, chosen for auditability (each bucket has a distinct

provenance rule, §3.2) over the more general key-value or embedding-retrieval memory these two systems use.

Evidence-based completion and orchestration (Surfaces: Verification/Policy). SWE-bench [9] operationalized "the task is done" as "the repository's own test suite passes after the patch is applied," rather than as a model's or human judge's opinion of a diff — the evaluation methodology STRAP's Verification surface generalizes into a protocol (run the cheapest deterministic check available before falling back to judgment). AgentBench [11] extended this evidence-based framing beyond single-repository code tasks to a broader suite of interactive environments, reinforcing that grounded, executable evidence — not a model's self-report — is the field's converging standard for judging agent task completion. AutoGen [10] and related multi-agent frameworks show that once more than one agent or role acts on a shared environment, explicit conversation/role protocols and human-in-the-loop checkpoints become necessary to keep the system coherent — the concurrency and policy concerns STRAP defers to its schema-only Policy and Tools surfaces (§3.5, §5) rather than solving inside the four fully-specified surfaces.

These eleven citations were selected because each documents a specific failure mode or design precedent STRAP's four fully-specified surfaces respond to directly; this is a targeted set grounding individual design decisions, not a survey of the broader agent-scaffold or agent-architecture literature (LangGraph-style graph orchestration, Voyager-style skill libraries, and design-by-contract programming are each relevant neighbors we do not attempt to cover here).

We use "harness" throughout to mean the non-model system that converts a task description into a verified environment change, and "skill" to mean a self-contained, model-agnostic instruction unit invoked with a defined input and returning a defined output, independent of any particular agent framework's terminology for the same concept.

3. The STRAP Protocol

STRAP specifies eight surfaces (Figure 1). Four — the surfaces that address the failure modes of §1 most directly — are given full data schemas and implemented as skills (§4); four are given a minimal schema so that a STRAP-conformant harness has a place to put them, without this paper prescribing an implementation, since they are inherently more environment-specific.

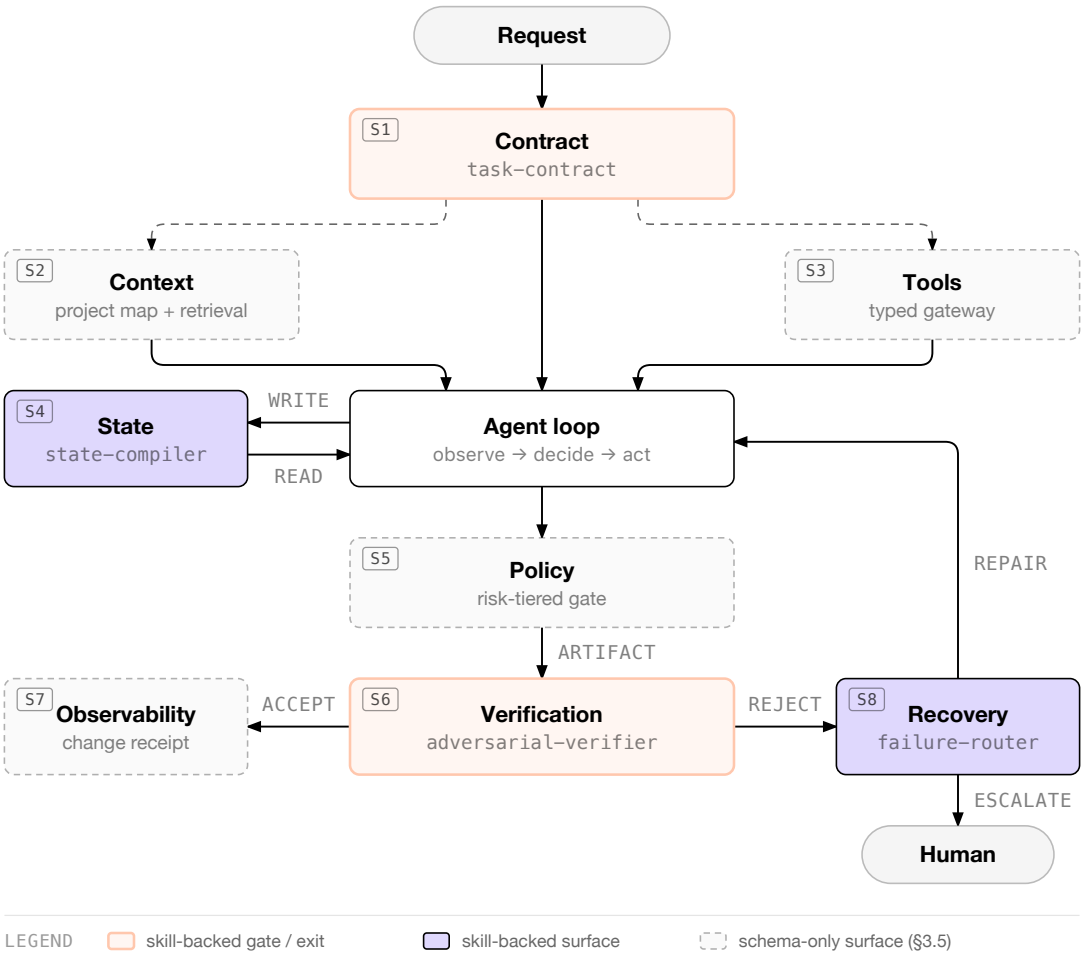

Figure 1: The STRAP loop: a request is gated by the Contract before any Context or Tool access, State is read and written on every step of the agent loop, and the loop ends only through Verification's accept path or Recovery's escalation path. Dashed boxes are the schema-only surfaces of §3.5.

3.1 Surface 1: Contract (fully specified)

A STRAP contract is a single YAML document (`TASK_CONTRACT.yaml` in the reference implementation) with six required top-level keys. It is the sole input authorizing an agent to act.

```yaml
contract_version: "1"
objective: <one sentence, the outcome that must exist>
scope:
  - <files, systems, or subsystems in bounds>
constraints:
  - <things that must not change>
acceptance:
  - id: acc-1
    statement: <a checkable statement, not an opinion>
```

```
approval_required:
  - <actions that must pause for a human, may be empty>
```

A request is not actionable under STRAP until this document exists and validates (six keys present, `acceptance` non-empty, every `acceptance` entry carrying a unique `id` and a `statement` phrased as a checkable predicate rather than a judgment — see the `task-contract` entry in §4). This is a hard gate, not a style preference: Verification and Recovery both take the contract as an input, and neither is meaningful without it. Verification (§3.3) checks the artifact against each `acceptance` entry's `statement`, keyed by its `id`; scope violations are checked against `scope` and `constraints`; and recovery escalation (§3.4) checks whether a proposed repair falls under `approval_required`.

Each `acceptance` entry's `id` is the identity a downstream consumer keys on, not the `statement` text. This is a deliberate, minimal answer to a specific interoperability failure: without a stable id, rephrasing an acceptance statement between the session that wrote the contract and the session that verifies it (or a second, independent STRAP implementation with its own wording conventions) silently breaks any consumer matching on the sentence itself. `contract_version` exists for the same reason at the document level — it is the field two independent implementations pin their parsers against, not a claim that STRAP's schema will change often.

3.2 Surface 2: State (fully specified)

STRAP state is a single JSON document (`STATE.json` in the reference implementation) with four required arrays (Figure 2), replacing transcript replay as the mechanism by which an agent recalls what it has already established.

```
{
  "facts": [{"id": "f-<session>-1", "text": "...", "source": "...", "as_of":
"ISO-8601", "superseded_by": null}],
  "decisions": [{"id": "d-<session>-1", "choice": "...", "reason": "...", "at":
"ISO-8601"}],
  "progress": {"completed": ["..."], "active": ["..."], "blocked": ["..."],
"remaining": ["..."]},
  "lessons": [{"text": "...", "trigger": "..."}]
}
```

Every `facts`, `decisions`, and `lessons`-bookkeeping `id` is namespaced by a short session identifier (`f-<session>-<n>`, `d-<session>-<n>`), not a bare sequence number — two independent sessions each starting their own counter at 1 would otherwise both produce `f-1` for unrelated facts, and a later merge of two per-session state files (§3.5′s Tools surface owns concurrent writes; per-session files are the mitigation for the case where two sessions *did* run concurrently) would either silently collide or require re-keying after the fact. Namespacing at generation time avoids the collision instead of resolving it after it has already happened. `acceptance[].id` in §3.1 does not need this: a contract is authored once, by one session, so `acc-1`, `acc-2`, … never has a second, independent writer to collide with.

`facts` are claims about the environment that were established by a tool result, not asserted by the model (a fact with no `source` is not a fact). `decisions` are choices the agent made where a plausible alternative existed, recorded with the reason, so a later session does not silently re-litigate them.

`progress` is the four-bucket task status. `lessons` are triggered rules learned from a failure (§3.4) — e.g. `{"text": "integration tests require TEST_DB_URL", "trigger": "before running tests"}` — consulted before the action that caused the original failure is attempted again. State is append-mostly: `facts` and `lessons` are only added to (`superseded_by`, below, is the one field of an existing fact a compiler may set); `progress` and `decisions` are the two fields a session is expected to mutate. `decisions` entries carry the same stable, never-reused `id` scheme as `facts` and `acceptance`, for the same reason: so a future consumer could reference a specific past decision without matching on the `choice` text, the same rewording problem `id`-keying solves for acceptance items in §3.1. This is forward-looking, not a claim about the current implementation — no shipped skill in §4 reads a `decisions[].id` today; the field exists so a `decisions`-referencing extension (e.g. a verifier that rejects an artifact because it contradicts a recorded decision) has a stable handle to cite, rather than needing a schema change to add one later. `facts.as_of` and `decisions.at` are deliberately different fields, not an inconsistency: a fact's `as_of` timestamp marks when the environment was observed to hold, and can be superseded by a later, contradicting observation (below); a decision's `at` timestamp marks a historical event — the moment a choice was made — which by definition cannot later be superseded, only recorded as a new, separate decision.

Append-only is not append-and-ignore: if a new tool result contradicts an existing `fact` about the same property of the environment, the compiler appends the new fact and sets the old fact's `superseded_by` to the new fact's `id`, rather than leaving two live, contradictory facts with no way to tell which a consuming session should trust. A fact with `superseded_by` set is history, not current state — a STRAP-conformant reader treats only facts with `superseded_by: null` as live. This does not solve concurrent writers to the same state file (§3.5′s Tools surface owns that problem); it only prevents the single-writer case where a long-running task's own state contradicts itself over time. Setting an existing fact's `superseded_by` pointer is itself a targeted mutation subject to that same single-writer assumption — two concurrent supersession attempts on the same fact is a genuine race, not merely a logical inconsistency, and is out of scope for the same reason.

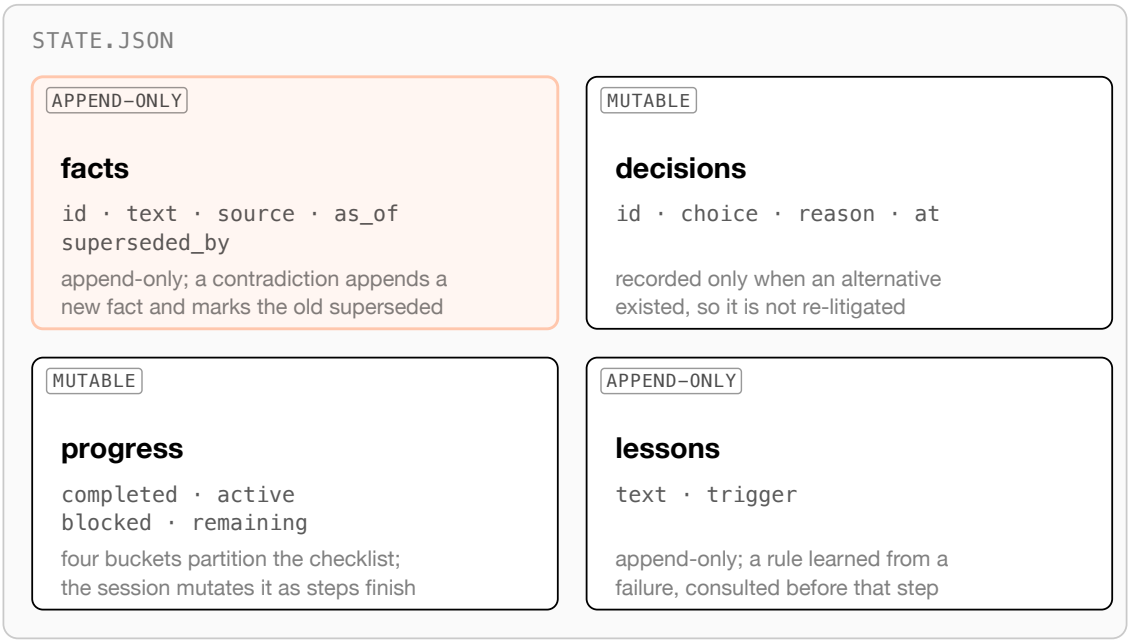

Figure 2: The four State buckets, their required shape, and the provenance rule that governs each one.

3.3 Surface 3: Verification (fully specified)

A STRAP verifier is a function `verify(contract, artifact, run_log) -> Verdict`, where `run_log` is the list of commands actually executed (used only as evidence of what was run, never as the worker's narrative) and `Verdict` is:

```
{
  "outcome": "accept | reject",
  "checked": [{"id": "acc-1", "result": "pass | fail | not-checkable"}],
  "rejection_reasons": [{"id": "acc-1", "reason": "..."}],
  "unverified": ["acc-2"]
}
```

The defining property is not the schema but the incentive: a STRAP verifier is instructed to attempt to reject, and is evaluated on whether it found a real defect a lenient pass would have missed, not on whether it agreed with the worker. It receives the contract and the artifact as separate inputs and is explicitly told not to reuse the worker's own reasoning trace as evidence, since a shared trace tends to reproduce the assumption that caused a mistake in the first place, matching the intrinsic self-correction failure mode reported by Huang et al. [5] and the self-preference bias reported by Zheng et al. [6]. Every `checked` entry must resolve through the cheapest available deterministic method before an LLM judgment is used for it — a test result or a schema check outranks a model's opinion that something "looks right" — and any acceptance item that could not be checked at all is listed in `unverified` by `id` rather than silently passed. `checked` and `rejection_reasons` key on

the contract's `acceptance[].id`, not on a re-typed copy of the `statement` text, so a later rewording of the statement never breaks the mapping between what was promised and what was checked.

3.4 Surface 4: Recovery (fully specified)

A STRAP recovery router is a function `route(failure) -> Action` (Figure 3), where `failure` is classified into one of a fixed, closed set of classes, each with a distinct default action:

Failure class	Signature	Default action
`timeout`	tool call exceeded its deadline	retry once with backoff
`invalid_arguments`	tool rejected malformed input	repair the call from the tool's own error, then retry
`missing_context`	agent lacked information to proceed	targeted retrieval of the specific missing source, not a full context reload
`verification_failure`	Adversarial Verifier rejected	inspect `rejection_reasons`, attempt one targeted repair
`permission_denied`	policy gate blocked the action	request approval, or select a lower-risk alternative in scope
`contradictory_requirements`	contract or environment is self-inconsistent	escalate to human, do not guess
`repeated_unchanged_failure`	same failure class recurs with no changed condition	stop the loop, escalate

Table 1. The seven-class failure taxonomy and each class's default recovery action (§3.4).

A router that cannot place a failure into one of the first six classes must place it in `repeated_unchanged_failure` and stop; STRAP treats an unclassifiable failure as equivalent to a repeat, since retrying an action whose failure mode is not understood changes nothing about the odds of success. Every retry action carries a budget (max attempts, max wall-clock time) set by the harness's Policy surface, not by the router itself — the router selects *what* to try next, not *how many times* it may be tried. (The reference skill ships small default caps for a harness that has no Policy surface yet.)

Escalation has two distinct targets, not one. By default — a `repeated_unchanged_failure` from any class other than `verification_failure`, or a single `verification_failure` on an acceptance `id` that has not failed before — escalation means "ask a human to unblock the environment or approve an action." The one exception: if the *same* acceptance `id` fails `verification_failure` twice in a row with a targeted repair attempted in between, the router escalates instead to "the contract's acceptance item may be wrong" and routes back to Surface 1 (Contract) for the `id` in question, not only to a human. A contract error and an environment error are different failures with different fixes, and a router that only ever escalates outward to a human conflates them — two

consecutive failures on one `id` is the router's signal that the fix belongs in the contract, not in the artifact.

This rule is keyed on the acceptance `id` recurring, never on the wording of the rejection matching across the two attempts — a verifier that describes the same underlying defect two different ways on its second pass still counts as the same repeat, since requiring exact text agreement would let a merely-reworded rejection loop indefinitely as if it were always a fresh `verification_failure`.

Detecting "twice in a row" needs somewhere to record the first failure, and STRAP does not add a router-specific field for this: the first `verification_failure` on an `id` is written to State's `lessons` (§3.2) as an ordinary learned rule, and a second `verification_failure` on the same `id` is a repeat exactly when a matching `lessons` entry already exists — reusing the one bucket in State already meant for failures that should change future behavior, rather than inventing new bookkeeping the schema doesn't otherwise define.

Recovery also does not assume Verification is installed: `verification_failure` can arrive with no `rejection_reasons` at all (a CI gate or a human reviewer classified it that way without producing that field), and the router still routes it as `verification_failure` using whatever failure description is available rather than stalling on a field that may never exist.

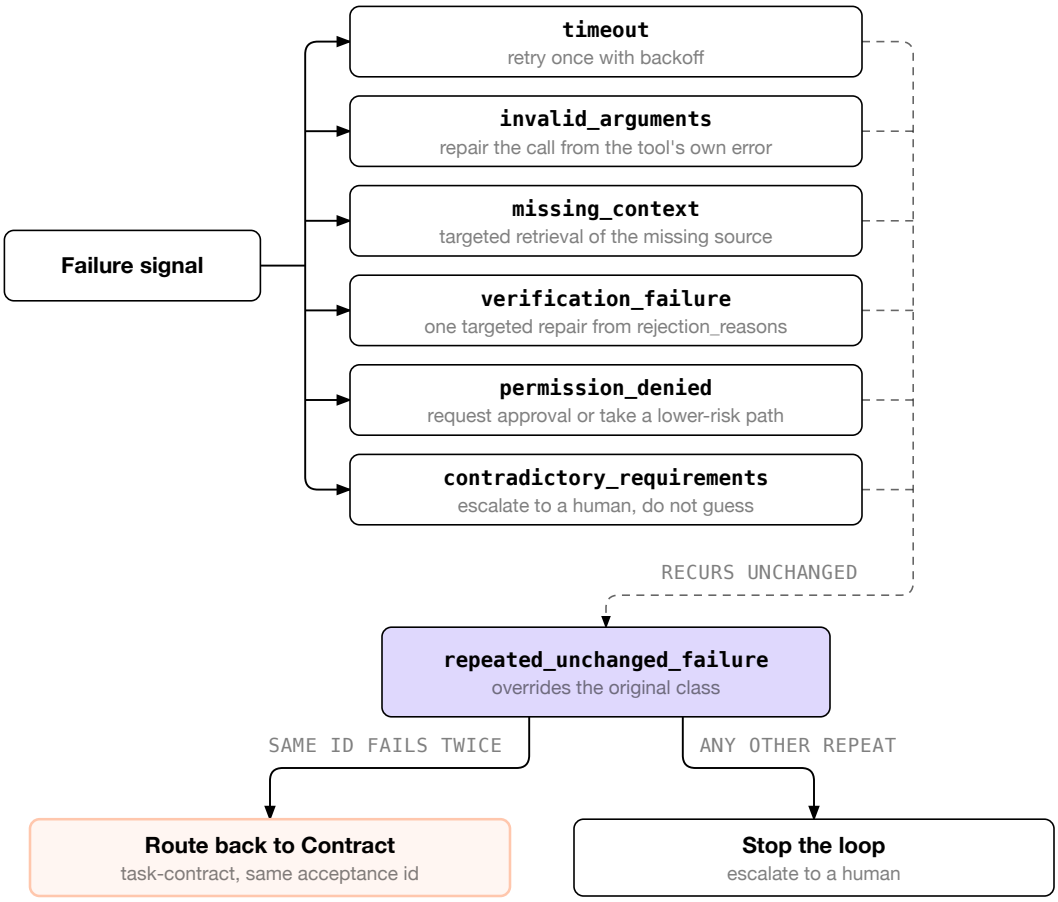

Figure 3: Six routed failure classes and their default actions, with the seventh, `repeated_unchanged_failure`, cutting across every class: a recurrence with no changed condition overrides the original classification and takes one of two escalation targets (§3.4).

3.5 Surfaces 5–8 (schema-only)

These four surfaces get a minimal shape, not a full specification, because their correct implementation is inherently environment-specific (§7 returns to why STRAP does not prescribe one). Each row is the complete requirement — nothing beyond the listed fields and readers is mandated:

Surface	Minimal shape	Who reads it
Context	a project map `{area: path}` loaded unconditionally, plus `retrieve(query) -> [source]` for on-demand detail; must be smaller than the full repository	the agent loop, on demand

Surface	Minimal shape	Who reads it
Tools	each tool declares `preconditions`, `success_evidence`, `failure_behavior`, and a `risk_class` drawn from Policy's tiers	the Tool Gateway, before executing any call
Policy	a four-tier risk table: `automatic`, `automatic_with_trace`, `approval_required`, `prohibited`	the Tool Gateway, Failure Router, and Verification, to resolve every tool/action to exactly one tier
Observability	a change receipt: `objective`, `changed`, `verified`, `not_verified`, `decisions`, `risks`, `approval_needed`	emitted once per run, derived mechanically from State and Verification — never re-authored by the agent from memory

4. Reference Implementation: Four Skills

The four fully-specified surfaces are implemented as four independent, installable skills. Throughout, the capitalized mechanism names of the abstract (Task Contract Generator, State Compiler, Adversarial Verifier, Failure Router) name the protocol roles, and the lowercase code names (`task-contract`, `state-compiler`, `adversarial-verifier`, `failure-router`) name the shipped skill files that implement them. Each is a plain-text instruction file (see `skills/` in the accompanying repository) with a declared trigger, a declared input, and a declared output, so that a harness can adopt one skill without the other three, or replace any one with a custom implementation that honors the same schema. "Without the other three" means each skill degrades rather than fails outright when a peer is missing — `failure-router`, for instance, does not require `adversarial-verifier` to be installed at all (§3.4) — but §5 still applies: adopting fewer than all four narrows which failure modes are covered, and a team should treat partial adoption as a deliberate scope decision, not a free lunch.

`task-contract` (§3.1). Trigger: an agent is about to act on a request that has no attached contract. Behavior: for any non-trivial task the skill refuses to let the agent proceed to tool use and instead produces the six-key YAML document of §3.1, asking the minimum necessary clarifying questions when `objective` or `acceptance` cannot be inferred, and never asking for anything inferable from the repository itself.

`state-compiler` (§3.2). Trigger: invoked at session end, at a context-window compaction boundary, or on demand. Input: the raw tool-call and message transcript since the last compilation. Output: an updated state JSON document, produced by extraction rather than summarization — a fact is

only added if a specific tool result supports it, and the skill is instructed to drop unsupported claims rather than soften them into hedged facts.

`adversarial-verifier` (§3.3). Trigger: a worker has produced a candidate artifact and claims an acceptance item is satisfied. Input: the contract and the artifact only (not the worker's reasoning transcript). Behavior: for each `acceptance` item, run the cheapest available check; for anything left to model judgment, actively search for a counterexample, a missed edge case, or an unsupported claim before agreeing, and report `unverified` rather than `pass` when no check exists.

`failure-router` (§3.4). Trigger: a tool call errors or the Adversarial Verifier returns `reject`. Input: the raw error or the verifier's `rejection_reasons`. Output: exactly one classification from Table 1 and the corresponding action; the skill is instructed to never emit a bare "try again" as its action.

These four skills compose in one loop (Figure 4): `task-contract` produces the authorization to act; `state-compiler` keeps the running record the router reads `lessons` from and a resumed session starts from; `adversarial-verifier` decides whether the loop can end; `failure-router` decides what happens when it cannot. No skill depends on a specific model, and none require the four schema-only surfaces to be present, though a production deployment should add at least a Policy risk table (§3.5) before granting an agent unattended write access to anything external.

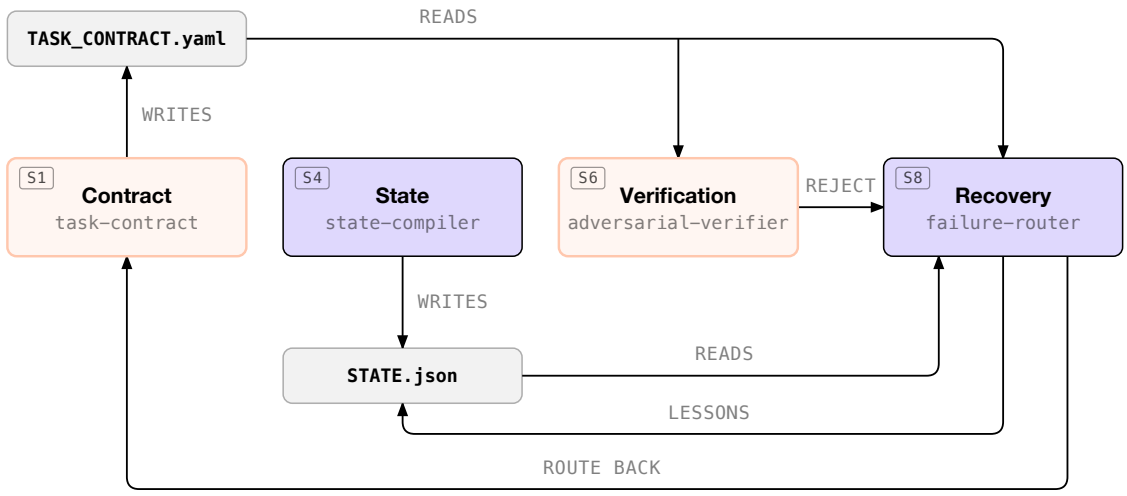

Figure 4: Data flow between the four shipped skills through two files: `task-contract` writes `TASK_CONTRACT.yaml`, which the verifier and the router read; `state-compiler` writes `STATE.json`, which the router reads and appends `lessons` to; a rejected verdict goes to the router, and a same-`id` repeat routes back to the contract (§3.4).

A skill is a prompt: instructions a model follows if it recognizes the trigger, and a model can fail to follow them under deadline pressure regardless of how the instruction is worded. §3.1′s "hard gate, not a style preference" is a design intent the four skills alone cannot fully deliver, since a plain-text instruction has no mechanism to physically stop a tool call. The reference implementation therefore ships two optional hooks alongside the skills: a `PreToolUse` hook for `task-contract` that blocks

side-effecting tool calls until `TASK_CONTRACT.yaml` exists in the working tree, and a `PreCompact`/`SessionEnd` hook for `state-compiler` that emits a reminder at the moment state would otherwise be silently lost, though — unlike the contract gate — a hook has no mechanism to force a skill invocation, only to make skipping one visible rather than silent. Both hooks are plain shell scripts kept in `skills/*/hooks/`. The contract gate ships with a runnable self-check, has been exercised in multiple live sessions, and — verified with `claude --debug hooks` logging — correctly blocks every single time it is actually invoked; it fails closed when it does run (it blocks rather than allows when `python3` is missing or the hook payload cannot be parsed), an availability trade-off an adopter should know about. **A real live-agent investigation also found that the *invocation itself* is not guaranteed**: Claude Code's own hook-dispatch layer was observed, on a small number of real live sessions, to skip invoking the hook entirely for a matched tool call, with no deterministic trigger identified despite directed attempts to reproduce one (full evidence and methodology in `docs/PREREGISTRATION_LIVE_AGENT_BENCHMARK.md`'s Run Log). This is a platform-level gap in Claude Code's dispatch, not a defect in this project's code, and is not fixable from inside a hook script — so the honest characterization of this mechanism is a strong, usually-reliable deterrent, not a proven, mathematically airtight gate, and §3.1′s "hard gate" language should be read with that qualification for the hook specifically (the schema-level requirement it enforces when it runs is unaffected). The compaction reminder has no self-check and has not been live-verified. Neither hook is required to use the skills, and both are optional precisely because their correct behavior depends on the exact hook I/O contract of the installed agent runtime, which STRAP does not standardize.

5. When Four Surfaces Are Enough

STRAP's four fully-specified surfaces are sufficient on their own when a task is bounded, single-session or a small number of resumed sessions, has no irreversible or externally-visible side effects, and where a human remains available to review the diff and the verdict before anything ships. This covers a large share of day-to-day coding-agent use: a scoped bug fix, a scoped refactor, a data-processing script run against a fixed input. In this regime, adding a full Context retrieval layer, a typed Tool Gateway, a Policy risk table, and a tracing/observability pipeline is very often needless weight relative to the failure the team actually has — complexity should be earned by an observed failure, not installed pre-emptively.

The four surfaces stop being sufficient once any of the following hold:

- the agent runs unattended across many sessions with no human review between them — Context and Observability become load-bearing, since there is no other memory or record;
- the agent can take an action with real external consequence (sending a message, spending money, deploying, deleting data) — Policy's risk gating becomes load-bearing, since Verification alone does not stop an action, only judges its result;
- the agent is one of several agents or tools operating on the same environment concurrently — Tools' preconditions and idempotency guarantees become load-bearing, since two uncoordinated writers are a race condition the four implemented surfaces have no mechanism to prevent.

A team that hits any of these should implement the remaining four schema-only surfaces from §3.5 before scaling usage, rather than compensating with a heavier prompt on the same four skills.

6. Empirical Evaluation of the Reference Mechanisms

The claims in §2 establish that the *problems* STRAP addresses — ungrounded tool use, intrinsic self-correction failing without external evidence, transcript-bound memory, and non-deterministic completion judgments — are documented in the literature. This section asks a narrower, answerable question about STRAP's own mechanisms: does the specific classification rule behind the Failure Router (Table 1) actually separate failure signatures correctly, and does routing a failure to a class-matched action actually resolve more of them than a policy that does not classify at all? Both experiments are small, fully deterministic or seeded, and reproducible from the scripts shipped in `experiments/` alongside this paper; neither involves a live LLM agent, and neither should be read as a benchmark of STRAP's skills as prompted text — that experiment is future work (§7).

6.1 Does the classification rule separate failure classes?

We implemented the reference decision table from §3.4 as a small rule engine: regex-based signature matching, plus a repeat check that fires when the same class recurs at the same step with unchanged arguments. We evaluated it against a hand-labeled corpus of 50 synthetic failure signatures spanning all seven classes, including six adversarial cases whose error text could plausibly match more than one pattern (e.g., "request timed out; server also returned 403 on retry"). The corpus was written to resemble real error-message conventions (HTTP status codes, Python exception names, curl/pytest/CI phrasing) rather than invented to fit the classifier's own patterns, and is roughly 2.5× the size of the corpus used in an earlier draft of this evaluation. Growing it surfaced a genuine test-harness bug: several unrelated corpus entries shared a generic step name (`plan`, `run_script`) and empty arguments, which spuriously triggered the repeat-detection rule against each other. The fix classifies each corpus entry independently and tests repeat detection with a dedicated pair (see `experiments/failure_classifier.py`).

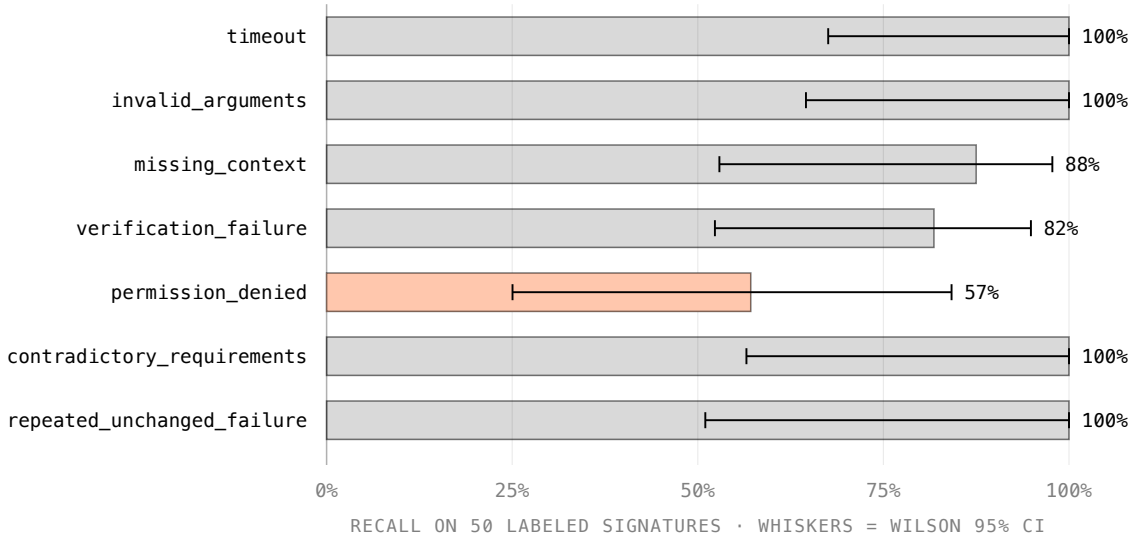

Figure 5: Recall of the reference failure-classification rule on 50 labeled synthetic failure signatures, one bar per STRAP failure class, with Wilson 95% confidence whiskers (per-class support is 4–11 cases, so the intervals are wide); overall accuracy 88.0%, Wilson 95% CI [76.2%, 94.4%].

The rule reached 88.0% overall accuracy (Figure 5; 44/50, Wilson 95% CI [76.2%, 94.4%] — reported because a bare point estimate on n=50 would overstate precision). The six errors split into two kinds. Three are genuine keyword gaps the larger corpus exposed: "assert 3 == 4" has no "assertion" or "expected…got" phrasing for the `verification_failure` pattern to match; "Access is denied" and "operation blocked…requires…approval" don't contain any of the `permission_denied` keywords. Three are ordering effects on the deliberately adversarial cases: a message containing both "permission denied" and "invalid argument" is caught by whichever pattern is checked first, not by semantic priority, and the same is true for a message mentioning both a timeout and a 404, and for a test failure whose message also carries a 403 status. This is a small, literal test of one reference implementation's regex coverage and ordering, not a claim about classification difficulty in general — but it is informative about the class of bug this design is prone to: a keyword-pattern classifier misses vocabulary it wasn't written with in mind and degrades on messages that legitimately contain two classes' vocabulary, and a production implementation should either broaden and reorder the patterns by specificity or replace the keyword matcher with a small trained or LLM-based classifier, keeping the same seven-class output contract.

We also ran the same classifier against a second, smaller corpus built to narrow the circularity threat along a different axis than corpus growth: 14 messages, each real, verbatim output from actually running a real command (`curl`, `ssh`, `git`, a real HTTP request to `api.github.com`, a real Python `unittest` run) rather than text written to resemble an error, labeled by a separate agent instance given only Table 1′s class signatures and the raw messages — never the classifier's source. On this corpus the same, unmodified classifier scored **10/14 = 71.4%, Wilson 95% CI [45.4%, 88.3%]** — lower than the 88.0% above, and we report it exactly as measured rather than adjusting the classifier

to close the gap. All four errors are the same failure mode: real tool phrasing the keyword lists don't cover — "fatal: not a git repository," "You must specify a repository to clone," "option … is unknown," and a bare `{"message": "Requires authentication"}` JSON body all fall through to the unclassifiable fallback, because none contain a literal keyword the patterns were written against.

A third, larger, and more independent run followed the pre-registered protocol in `docs/PREREGISTRATION_INDEPENDENT_CORPUS.md` (two amendments logged there before any data was touched: N reduced from the registered 100 to 30 for session-time feasibility, and labeling performed by a second AI agent instance rather than the registered independent human, since none was available). Messages were sourced mechanically from real, currently-failing GitHub Actions jobs on ten real public repositories (pytest, numpy, pandas, django, flask, sqlalchemy, scikit-learn, fastapi, requests, httpx) via the GitHub API, pooled (290 raw extractions), deduplicated to 74 unique messages, and 30 sampled with a fixed seed — not the first 30 encountered, and not hand-picked. The blind labeler marked 10 of 30 `insufficient_info` (mostly truncated build-tool fragments the mechanical extraction pulled with no usable error text attached) and these were excluded per the registration's own rule, leaving 20 evaluated: the same, unmodified classifier scored **9/20 = 45.0%, Wilson 95% CI [25.8%, 65.8%]** — markedly lower than both prior corpora. Ten of the eleven errors are real `verification_failure` messages misclassified as the unclassifiable fallback, because most real pytest `FAILED path::test - ExceptionType: message` summary lines simply do not contain the literal words "assert" or "expected…got" the pattern looks for — a specific, actionable regex-coverage gap this corpus surfaced that the smaller ones did not. Full collection and labeling methodology, including a real extraction-rule bug found and fixed before any labeling began, is recorded in the pre-registration's Run Log.

This progression — 88.0% (self-authored) → 71.4% (n=14, real-sourced) → 45.0% (n=20, real-sourced at larger scale, independently sourced) — is reported in full, not just the most favorable point, because it is the whole point of running these corpora at all: a classifier tuned against self-authored synthetic phrasing does not automatically generalize to how real tools actually word their errors, and each successive, more independent corpus made that gap larger and clearer, not smaller.

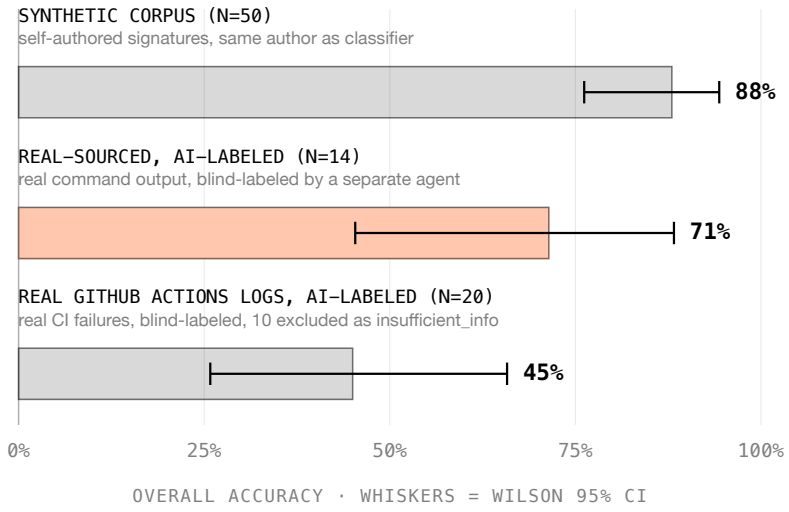

Figure 6: The same, unmodified classifier scores 88.0% on the self-authored synthetic corpus (n=50), 71.4% on a small real-sourced corpus (n=14), and 45.0% on a larger, independently-sourced-and-labeled corpus of real GitHub Actions failures (n=20 evaluated) – reported as measured, with Wilson 95% intervals, not adjusted to close the gap.

6.2 Does routing a failure to a matched action resolve more of them?

We built a Monte Carlo simulation (seed 20260907) of a bounded recovery loop over a synthetic environment with six failure reasons drawn uniformly per episode: five recoverable ones, each with a known, fixed probability (0.55–0.95) that its *correct* fix action resolves it in one attempt, reflecting that not even the right fix always works the first time, and a sixth, `contradictory_requirements`, that no automated action resolves. (`repeated_unchanged_failure` is not a simulated reason because it is an outcome of the loop, not an input to it.) Every episode has a budget of 4 attempts; this is an assumption of the experiment, since §3.4 delegates budgets to the Policy surface. We compared three policies for 20,000 simulated episodes each:

- **naive retry** repeats the original failing action without classifying anything. It is credited a 12% baseline chance of success per attempt (`FLAKE_P`) to reflect ordinary transient flakiness (a dropped connection, a race) rather than being rigged to zero. The baseline applies to every reason, including `contradictory_requirements`, which favors this baseline.
- **ground-truth routing** applies the reason's correct fix action with perfect classification and stops (escalates) once the same failure class recurs twice with no change, per the `repeated_unchanged_failure` rule — a ceiling on what routing can buy. Because of that rule, both routing policies stop after two unchanged failures, so their effective budget is 2, not 4.
- **realistic routing** is identical to ground-truth routing except that whether it acts on the correct class each episode is drawn from that class's *actual measured recall* in §6.1, so a real classification error applies the wrong fix action for the whole episode, exactly as it would in a deployed router that

trusts its own classification. A misrouted episode is credited the same 12% per-attempt flakiness as naive retry, but under the repeat rule it gets two attempts rather than four — an asymmetry that works against routing.

This composes §6.1′s measured error rate into §6.2′s simulation rather than reporting the two experiments side by side with no combined number.

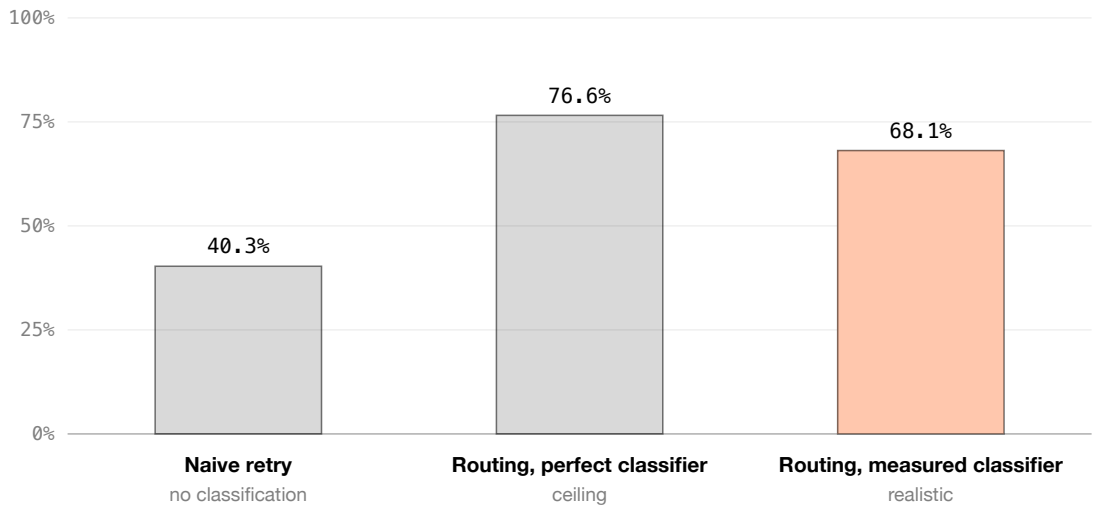

Figure 7: Overall resolution rate over 20,000 simulated recovery episodes per policy, comparing naive retry against a perfect-classification ceiling and a realistic-classifier-driven routing policy, all under a 4-attempt budget. Wilson 95% intervals are ±0.7 percentage points or narrower and are omitted from the bars.

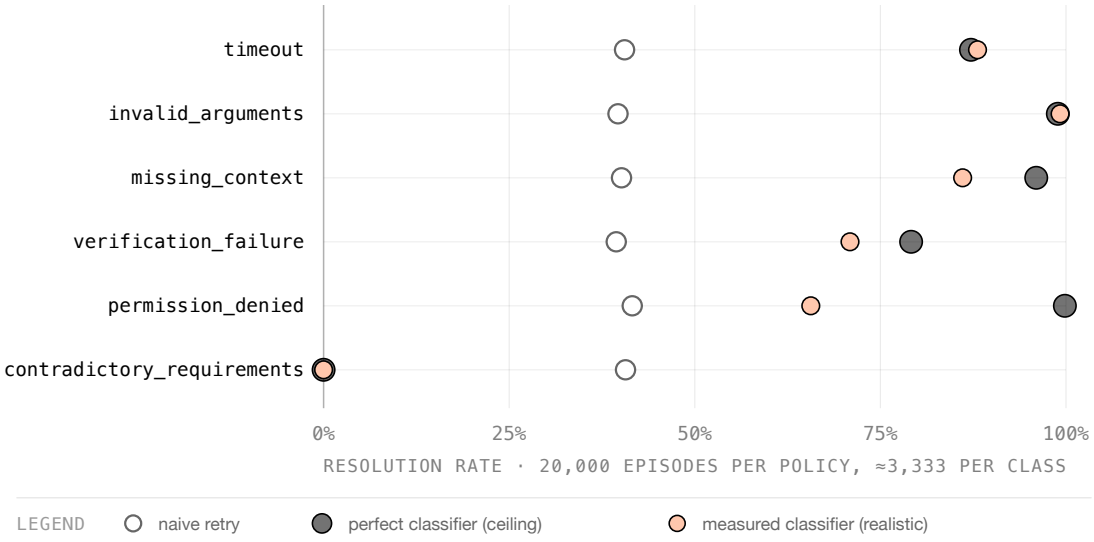

Figure 8: Resolution rate by failure class across the same three policies (20,000 episodes per policy, about 3,300 per class). Hollow marks are naive retry, filled ink marks the ceiling, coral marks realistic routing; where the coral mark covers the ink one, the two policies are indistinguishable within sampling noise.

Overall, naive retry resolved 40.3% of episodes (Wilson 95% CI [39.6%, 41.0%]), ground-truth routing (the ceiling) 76.6% [76.0%, 77.2%], and realistic routing — §6.1′s actual per-class recall in place of perfect classification — 68.1% [67.5%, 68.8%] (Figure 7). Routing clears naive retry by a wide margin even with real classification error folded in, giving back about a quarter of the ceiling's advantage to that error. The per-class breakdown (Table 2, Figure 8) shows where: the drop from ceiling to realistic tracks §6.1′s measured recall. It is nil for the two classes with 100% recall, where the two policies are the same process and their figures differ only by sampling noise (which is why realistic can nominally exceed the ceiling there), and largest by far for `permission_denied`, the class with the weakest recall. The Monte Carlo intervals above are narrow; the uncertainty that matters for the realistic figure is the classifier-recall uncertainty it inherits from §6.1′s small per-class supports (Table 2, last column).

Failure class	Naive retry	Ceiling	Realistic	§6.1 recall (95% CI)
`timeout`	40.5%	87.2%	88.1%	100% [68%, 100%]
`invalid_arguments`	39.7%	98.9%	99.2%	100% [65%, 100%]
`missing_context`	40.1%	96.0%	86.1%	88% [53%, 98%]
`verification_failure`	39.4%	79.1%	70.9%	82% [52%, 95%]

Failure class	Naive retry	Ceiling	Realistic	§6.1 recall (95% CI)
`permission_denied`	41.6%	99.9%	65.6%	57% [25%, 84%]
`contradictory_requirements`	40.7%	0%	0%	100% [57%, 100%]

Table 2. Resolution rate by failure class across the three policies (20,000 episodes per policy, about 3,300 per class), alongside each class's classification recall from §6.1 with its Wilson 95% interval. Both routing policies resolve 0% of `contradictory_requirements`: the ceiling by construction (§3.4 specifies escalation, not resolution, for that class), and realistic routing because its measured recall on that class is 100%, so it never misroutes it into a retry — the expected floor, not a defect. Naive retry's 40.7% on that row is the flakiness baseline, which applies to every class.

Because naive retry's headline number depends entirely on the one flakiness constant it is credited with, we swept it from 5% to 40% instead of reporting only 12% (Figure 9; N=5,000 per point, Wilson 95% intervals ±1.4 points or narrower). A blind retry with a budget of 4 has the closed form $1 - (1 - \texttt{FLAKE\_P})^4$, which the simulated points track. The comparison is sensitive to this constant, and 12% should be read as a stated, moderate assumption, not a measured rate — no log of real per-attempt retry-success frequencies backs it. Neither routing policy survives the whole swept range: interpolating the sweep puts naive retry's crossover with realistic routing (68.1%) at roughly **FLAKE_P ≈ 0.245** and with the ceiling (76.6%) at roughly **FLAKE_P ≈ 0.30**, and at 0.40 naive retry resolves 86.7%, above both. Take from this that routing's advantage over blind retry is robust while a non-classifying policy's flakiness stays below roughly one attempt in four succeeding on luck alone, and is gone above roughly one in three — whether real tool failures typically fall below or above that line is an open empirical question this paper does not answer, and the 12% figure used elsewhere is a stated assumption inside the safer range, not a claim the true rate sits there.

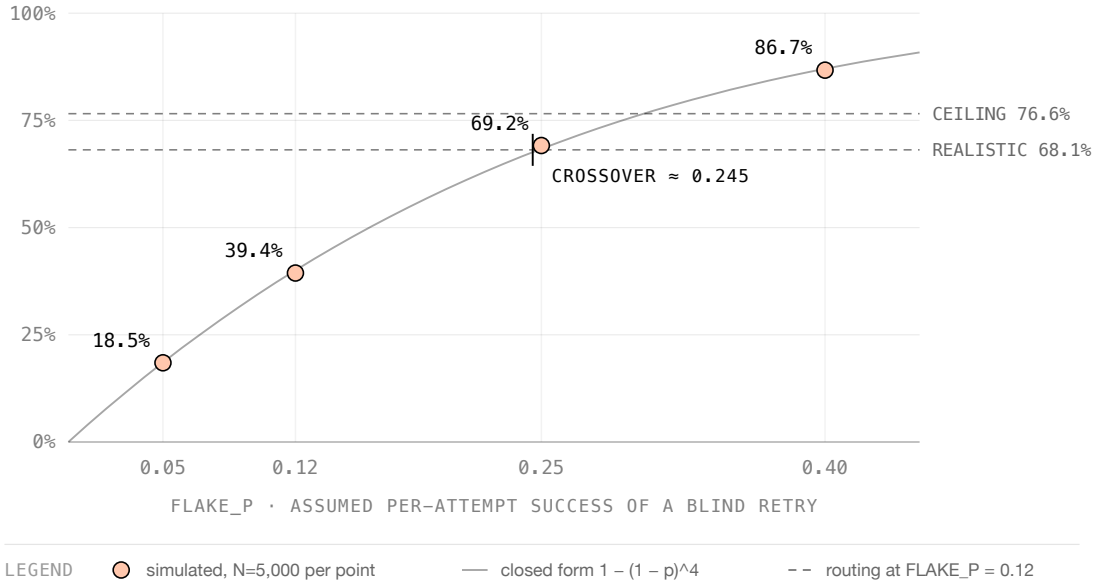

Figure 9: Naive-retry resolution rate as the assumed per-attempt flakiness constant varies from 5% to 40%, holding the 4-attempt budget fixed: simulated points (N=5,000 each) over the closed-form curve, against the two routing policies' rates at the stated 12% assumption. Naive retry overtakes realistic routing at about 0.245 and the ceiling at about 0.30.

7. Threats to Validity and Scope

§6 already states what its experiments do and do not show; this section adds the threats that discussion doesn't cover. The claims in §2 about intrinsic self-correction and judge self-preference are drawn from prior published work on other systems [3–6], not from a new experiment run on STRAP itself — treat them as motivating evidence for the design, not as validation of this specific artifact. A pre-registered comparison of a STRAP-harnessed agent (the four skills plus both enforcement hooks, installed exactly as shipped) against the same agent with only a system prompt, on a task bank randomly sampled from an existing third-party benchmark, would close that gap. That comparison is pre-registered — hypothesis, task-sampling procedure, sample size and power calculation, stopping rule, exclusion rules, and analysis plan fixed in `docs/PREREGISTRATION_LIVE_AGENT_BENCHMARK.md` — and was attempted at a small, logged-amendment pilot scope (1 of 6 registered pilot tasks; the confirmatory N=94 remains unrun); a null or negative result is registered as a reportable outcome, not a failure of the study, and what the pilot actually found is reported below.

§6.2′s figures carry Wilson intervals, but those intervals bound only Monte Carlo sampling noise. The realistic-routing figure also inherits the uncertainty of the per-class recall estimates that drive it, which rest on 4–11 labeled cases per class (Table 2); a reader should treat 68.1% as conditional on those recall point estimates, not as a figure with ±0.7-point precision in its own right.

§1′s cross-implementation classification comparison (81.2% vs. 88.0% on the same corpus, two independently-designed classifiers) demonstrates that STRAP's schemas support a second imple-

mentation well enough to run a real, reproducible comparison — it does not demonstrate that two implementations converge on the same decision in general, only that they land in the same neighborhood on this one fixed, 48–50-case corpus. Neither classifier in that comparison uses a model; the comparison says nothing about whether two independent LLM-driven implementations of the `adversarial-verifier`/`failure-router` skills would agree with each other, which remains untested and is a natural extension of the still-open live-agent-benchmark item above.

§6.1′s corpus carries a further, distinct threat beyond its size: both the failure signatures and the classifier's regex patterns were authored by the same person, rather than the corpus being sourced from real production error logs or labeled independently of the implementation. Growing the corpus from 19 to 50 cases and writing the additions around realistic tool/HTTP/CLI error conventions (§6.1) narrows this threat along one axis (message realism) — it surfaced three genuine keyword gaps the smaller corpus didn't exercise. The 14-case real-sourced, blind-labeled corpus (§6.1) narrows it along the other axis (labeling independence) and is the stronger of the two mitigations, though still not a full resolution: message *selection* (which commands to run) was still the same person's choice, only *labeling* was blind to the implementation. Both mitigations together still fall short of the clean version of this control — a corpus collected from real production error logs, at production scale, and labeled by an independent human with no relationship to the classifier's author. That study is pre-registered in `docs/PREREGISTRATION_INDEPENDENT_CORPUS.md` (source-sampling procedure, target N=100 with the interval-width justification for that figure, labeler-independence requirement, inter-rater reliability plan, analysis plan) and was run at reduced scope under two logged amendments — N=30 rather than 100, and a second AI labeler rather than the registered independent human — producing the 45.0% (n=20, Wilson 95% CI [25.8%, 65.8%]) figure above. The confirmatory version, at full N with a genuine human labeler, remains unrun. A reader should treat 88.0% as an upper bound reflecting the corpus's own construction, 71.4% (n=14) and 45.0% (n=20) as two independent, more honest estimates of real-world generalization at increasing scale and source-independence, and the clear downward trend across all three as itself informative — not an artifact of any one corpus's noise, since each corpus was built to narrow the threat along a different, disclosed axis and each came in lower than the last.

The live-agent benchmark (`docs/PREREGISTRATION_LIVE_AGENT_BENCHMARK.md`) was also attempted at reduced scope — 1 of the pre-registered 6-task pilot, not the confirmatory N=94 — and is reported in that document's Run Log rather than here, because what it found was not a resolution-rate result at any usable sample size: a real, previously-unknown bootstrap deadlock in `require-contract.sh` (fixed and regression-tested, 10/10 self-check cases passing, during this same run), and a separate, still-unexplained intermittency in whether the hook fires at all, observed once and not yet root-caused. Both the N=94 confirmatory study and the hook-reliability question remain open.

The protocol is also deliberately narrow: it takes no position on multi-agent orchestration, on which model to use, on memory retrieval architecture, or on how Context and Tools should be implemented beyond the schema in §3.5, because those choices are highly environment-specific and a protocol that mandated an answer for all of them would be exactly the kind of pre-emptive complexity §5 warns against.

8. Conclusion

Agent reliability work has, correctly, moved its attention from the model to the system around the model. What that system needs is not a new framework but four small, checkable artifacts: a contract before action, durable state instead of transcript replay, a verifier incentivized to reject, and a router that treats failure as data rather than noise. STRAP specifies those four artifacts as literal schemas and ships them as four independent, model-agnostic skills. §6 and §7 report what has and has not been shown about them; the honest remainder is that a live agent running these four skills, on real tasks, against a real baseline, has not yet been measured — the classification rule and the routing policy have, and each of those results is reproducible from the scripts this paper ships alongside it.

Colophon

Code, skills, hooks, and this paper's text and figures are released under the MIT License (see the repository's `LICENSE` file); the four `SKILL.md` files and both hook scripts carry their own SPDX header so the license travels with them when copied out of the repository, per the install instructions in §4. This paper was drafted with AI assistance and reviewed and approved by the author.

References

[1] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). ReAct: Synergizing Reasoning and Acting in Language Models. *ICLR 2023*. arXiv:2210.03629.

[2] Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). Toolformer: Language Models Can Teach Themselves to Use Tools. *NeurIPS 2023*. arXiv:2302.04761.

[3] Madaan, A., Tandon, N., Gupta, P., et al. (2023). Self-Refine: Iterative Refinement with Self-Feedback. *NeurIPS 2023*. arXiv:2303.17651.

[4] Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K., & Yao, S. (2023). Reflexion: Language Agents with Verbal Reinforcement Learning. *NeurIPS 2023*. arXiv:2303.11366.

[5] Huang, J., Chen, X., Mishra, S., Zheng, H. S., Yu, A. W., Song, X., & Zhou, D. (2024). Large Language Models Cannot Self-Correct Reasoning Yet. *ICLR 2024*. arXiv:2310.01798.

[6] Zheng, L., Chiang, W.-L., Sheng, Y., et al. (2023). Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. *NeurIPS 2023 Datasets and Benchmarks Track*. arXiv:2306.05685.

[7] Packer, C., Fang, V., Patil, S. G., Lin, K., Wooders, S., & Gonzalez, J. E. (2023). MemGPT: Towards LLMs as Operating Systems. arXiv:2310.08560.

[8] Park, J. S., O'Brien, J., Cai, C. J., Morris, M. R., Liang, P., & Bernstein, M. S. (2023). Generative Agents: Interactive Simulacra of Human Behavior. *UIST 2023*. arXiv:2304.03442.

[9] Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. (2024). SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *ICLR 2024*. arXiv:2310.06770.

[10] Wu, Q., Bansal, G., Zhang, J., et al. (2023). AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation Framework. arXiv:2308.08155.

[11] Liu, X., Yu, H., Zhang, H., et al. (2023). AgentBench: Evaluating LLMs as Agents. *ICLR 2024*. arXiv:2308.03688.