

Brevet: Change Control for What Self-Evolving AI Agents Learn

A governed evolution loop that makes agent learning promotable, auditable, and revocable

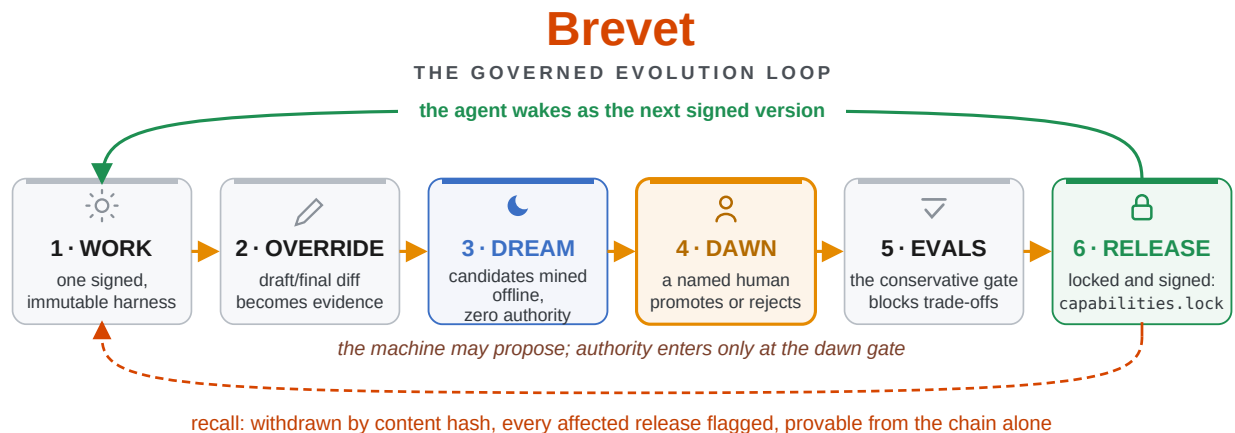
Arsalan Shahid, Gordon Suttie, and Philip Black
Brightbeam AI

arsalan.shahid@brightbeam.com

September 2026

Abstract

Deployed AI agents can change their behaviour without changing model weights. They store memories, accumulate skills, and revise prompts. These changes leave operators with three questions. What has the agent learned, who authorised each capability, and how can an incorrect capability be withdrawn? We present BREVET, a governed evolution loop with a reference runtime for placing learned agent capabilities under change control. Experts' corrections become structured overrides. An offline mining cycle turns recurring divergences into candidate capabilities with no operational authority. Human review determines which candidates may advance, evaluations test proposed changes, and release records associate approved capabilities with versioned inventories. Recall records withdrawal and identifies affected releases. The design connects the Collaborative Human-Agent Protocol's evidence model with the governed-memory principles used for tacit fragments captured by Metis. A deterministic demonstration and a conversational-assistant integration illustrate the lifecycle. We also specify a governed-adaptation benchmark that evaluates improvement, regression discipline, lineage completeness, and recall compliance together. The reference implementation for Brevet is available at <https://github.com/BrightbeamAI/brevet>.



Agents propose deltas; evidence tests them; humans promote them; the runtime only ever executes signed versions.

1 Introduction

Although recursive self-improvement in AI often centres on model weights, a more immediate operational form concerns the software harness surrounding those models. An agent with persistent memory can change its behaviour each time it decides something is worth remembering [1, 2]. An agent with a skill library changes its behaviour each time it writes itself a new skill [3]. Prompt-optimisation systems rewrite the instructions an agent runs on, using feedback from its own outputs [4, 5]. None of these mechanisms requires retraining the underlying model, yet each can change what the system does. The agent an organisation evaluated at deployment may therefore differ from the agent it runs a month later.

Consider a concrete, illustrative workflow. A quality reviewer in a pharmaceutical plant receives an agent’s draft triage for a process deviation. A pump showed high vibration during a clean-in-place (CIP) run, and the agent drafts *severity: minor, log and monitor*. She recognises the vibration as a possible sign of seal wear under these operating conditions, so she finalises the record as *major* and moves on. She does the same the following week, and the week after. The agent’s memory picks up the pattern, and its drafts begin to match her judgement. This appears to be the desired outcome. But no one wrote the change down. No one with authority approved it. There is no version of the agent from before the change to compare against or return to. When the rule later turns out to over-generalise because the vibration was a sensor artefact on one pump family, there is no discrete rule to revoke. Repeated across the memories, skills, and prompt edits that a fleet accumulates, this pattern raises three questions.

1. *What does it know?* Not the weights, but the accumulated rules, skills, memories, and tool bindings acquired since deployment.
2. *Who approved that?* A learned behaviour changed a real decision. Which accountable person said it could ship, and on what evidence?
3. *How is it taken back?* The learned rule is wrong. Where does it live, which versions shipped it, and what evidence shows that it is no longer in use?

The technical scope of self-improvement is also expanding. A *harness* is the surrounding software that selects prompts, manages context, invokes tools, and controls an agent’s execution. The Darwin Gödel Machine iteratively modifies agent code and evaluates descendants on coding tasks [6]. Self-Harness mines execution failures and validates proposed harness edits [7]. Recursive Harness Self-Improvement represents the agent loop as a prompt-level specification and revises it using feedback on earlier versions [8]. Improvement is recursive when an updated system contributes to subsequent updates. Together, these studies extend the scope of adaptation from individual prompts and skills to the mechanisms that organise an agent’s execution, with progress assessed under their respective task and evaluation conditions.

The learning environment can evolve as well. POET jointly generates environmental challenges and optimises agents to solve them [9]. More recently, EvoEnv constructs executable reasoning environments that generate tasks and verify responses [10]. These developments broaden the scope of governance. As agents, harnesses, and learning environments evolve, operators need to track changes to both operational behaviour and the evaluators that supply evidence of improvement. This creates a need for an accountable boundary between proposing a change, assessing its effects, and authorising its use. The framework presented in this paper addresses that boundary through explicit capability records, human review, versioned releases, and recall.

Our previous work provides two foundations. CHAP represents human–agent collaboration through structured tasks, artefacts, reviews, overrides, and evidence records [11]. It makes the record of work explicit, but does not by itself determine which lessons should change a deployed agent. The governed-memory model for tacit fragments, operationalised through Metis, addresses how situated expert judgement can be

captured with provenance, applicability conditions, and authority [12]. A tacit fragment captured by Metis is a bounded record of practical knowledge, not a general permission to act. This paper builds on these foundations by extending governed memory to the lifecycle of learned agent capabilities. CHAP supplies the collaboration evidence, Metis supplies a process for capturing and governing tacit knowledge, and the framework developed here connects recurring operational corrections to reviewed candidates, evaluated releases, and recall. Responsibilities follow this division of function. The framework maintains approval decisions, capability state, and recall records; CHAP records collaboration and review; Metis captures tacit fragments.

For high-consequence work that depends on expert judgement and organisational responsibility, we adopt accountable human approval as a requirement for changes to an agent’s operational authority. Pharmaceutical quality guidance and electronic-record controls [13, 14], the EU AI Act [15], and the NIST AI RMF [16] motivate traceability, oversight, and lifecycle risk management within their respective scopes. Applying these principles to agent adaptation requires a process that connects each proposed change to supporting evidence, a named approval decision, and a controlled release. Organisations also need their agents to improve, with evidence that each change is worthwhile. The governing objective is therefore to make learning *promotable, auditable, and revocable*.

Evaluation alone does not provide this control. Benchmarks can be rerun after deployment, but a score does not identify who approved a behavioural change or how to withdraw it. Capability and memory benchmarks [17, 18] answer important performance questions; governing successive versions requires additional lifecycle records. The controls are familiar. Software engineering pins dependencies in lockfiles and publishes bills of materials [19]. Supply-chain security signs artefacts and verifies provenance [20–22]. Quality-managed workplaces use change requests, sign-off, versioning, and rollback for procedures. This boundary calls for explicit capability records, human review, versioned releases, and recall.

This paper presents BREVET, a framework for governed adaptation with a local-first Python reference runtime that wraps the agent framework an organisation already uses. Its *governed evolution loop* follows one design principle.

Agents propose deltas; evidence tests them; humans promote them; the runtime only ever executes signed versions.

The paper makes five contributions.

1. *A model of governed agent adaptation* (Section 3), comprising a single *capability object* for learned artefacts and a three-layer authority model. Conditional arguments state when approval-before-use and recall exclusion follow from the design.
2. *A stage-by-stage account of the loop* (Section 4), covering work, override, dream, dawn, evals, release, and recall. Each stage is motivated by what goes wrong without it and illustrated on the running example. The dream stage develops the endogenous capture pathway specified in the governed-memory model of Shahid et al. [12].
3. *A reference implementation* (Section 5) with adapters for common agent invocation patterns, shared lifecycle interfaces, and a hash-linked evidence chain compatible with CHAP.
4. *A worked demonstration* (Section 6) of the lifecycle in synthetic pharmaceutical deviation triage, replayable offline, and a conversational-assistant case study (Section 7). Numerical analysis characterises the release gate and quantifies how sample size affects the interpretation of regression evidence.
5. *The design of a governed-adaptation benchmark* (Section 8) that reports improvement, regression discipline, lineage completeness, and recall compliance as a profile rather than a single number.

Brevet provides a governance layer for existing agent frameworks, organising adaptation around accountable human approval, explicit capability inventories, and traceable release and recall decisions. This approach connects the evidence of what an agent has learned to the authority under which that learning may influence its behaviour. The resulting research questions concern both effectiveness and governance. How much can the loop improve task outcomes, how reliably can it control regressions and enforce withdrawal, and how much reviewer attention does it require? Section 8 develops a benchmark for evaluating task outcomes and governance properties, complemented by field studies to assess reviewer effort.

2 The Problem of Ungoverned Learning

2.1 Self-improvement moved up the stack

The mechanisms by which deployed agents change themselves are concrete and increasingly varied. Persistent memory systems decide what to store and what to retrieve, and stored items steer future behaviour [1, 2]. Skill libraries accumulate reusable executable procedures; Voyager’s agent generates, verifies, and stores its own skills [3]. Reflection loops convert failure into revised strategy [23, 24]. Prompt and harness optimisers rewrite instructions, few-shot examples, and control flow against a metric [4, 5], and recent work optimises the surrounding harness as a whole [7]. Each mechanism can support useful adaptation. Collectively they mean that the unit demanding governance is not only the model but also the *harness*, comprising the prompts, tool bindings, loop policy, memory bindings, and safety controls that make a model an agent. The harness is therefore an important location of deployed learning, even when model weights remain fixed. If its changing components are not versioned together, a model version alone does not identify the running agent.

Naive accumulation can also degrade performance. A memory store without curation can introduce irrelevant or conflicting material into future contexts, and an agent that remembers everything may have to reason over an ever-larger body of material. Storage is cheap, but judgement about what deserves to steer behaviour is not. What matters is not whether an agent stores things but which stored things get promoted into use, on what evidence, and under whose authority. Each new memory can change the handling of unrelated tasks, for better or worse. This creates a form of behavioural interference as knowledge accumulates across tasks. The practical concern parallels interference in sequential learning [25], while arising here through changes to retrieved context and instructions with model parameters held fixed.

2.2 Static evaluation is not change control

The standard response, evaluate the agent, does not answer the three questions of Section 1, for two reasons. First, benchmarks require curated tasks and evaluation criteria. Judgement-heavy work often lacks an inexpensive, reliable executable oracle for questions such as *was this deviation major?* A human decision may supply a label, but that label can itself require review. Second, repeated evaluation is not change control. A benchmark score describes a system frozen at a moment; a learning agent is a *lineage* of systems, one version after another, each differing from the last by what it learned. Governing a lineage requires runtime machinery, including an inventory of what each version knows, an approval record for each change, and a mechanism that removes a change from the lineage with verifiable records and evidence of enforcement. Useful analogues come from software supply chains, where lockfiles, signed provenance, and advisories govern lineages of artefacts [19–21].

2.3 The standard organisations already apply to people

The controls needed for agent learning resemble those organisations apply when people change shared working procedures. A new hire learns fast, and changes they propose to those procedures are governed.

When they think a standard operating procedure should change, they write a change request. A supervisor signs it off by name. The procedure moves from version twelve to version thirteen, and the history is kept. When a change proves unsuitable, it is rolled back. These records allow trust to extend beyond personal acquaintance.

An agent without comparable controls may rewrite its operating rules through memory, skills, and prompt edits without creating a discrete change for anyone to approve. Without versions and diffs, yesterday's configuration may be difficult to recover. Without an inventory, an operator may not know where a mistaken rule persists. The analogy concerns how organisations authorise changes to shared working procedures. Brevet applies comparable change control to learned agent capabilities, with review effort treated as an operational cost to be measured alongside improvements in task performance.

2.4 Requirements

From the three questions, the policy assumption, and the analogy, we derive six requirements for a governed adaptation runtime. Each maps to a stage of the loop in Section 4.

- R1. Attributable approval.* Every increase in a learned capability's authority is a decision by a named human or an accountable human group, recorded with evidence.
- R2. Immutable execution.* The running agent executes exactly one signed version of its harness and cannot modify its own harness, inventory, or authority state.
- R3. Capability inventory.* Every release enumerates what the agent has been authorised to use, including each learned capability, its content hash, its provenance, and its approver, in one machine-checkable artefact.
- R4. Regression discipline.* No change ships on net improvement alone. Review must identify regressions in protected tasks and risk groups rather than hiding them behind gains elsewhere.
- R5. Revocability with proof.* Any capability can be withdrawn such that withdrawn content cannot re-enter a release while its recall remains in force. Every affected release must be identified, and enforcement must be evidenced where the capability was used.
- R6. Consent and provenance.* Learning derived from human work respects declared consent scope, and withdrawal of consent is honoured through the same revocation mechanism, never by silent deletion.

3 Objects and Authority

Before describing the loop, we define the two structures it governs. The first is the object that represents a piece of learning; the second is the ladder that says how much that object is allowed to do. The vocabulary extends the governed-memory model introduced for tacit knowledge in Shahid et al. [12] from memory to everything an agent learns. That model admits two capture pathways. In the *exogenous* pathway, experts contribute knowledge through structured elicitation. In the *endogenous* pathway, candidate capability is inferred from the record of work itself. The model defines the governance contract both pathways must satisfy, but its endogenous pathway is specified as a model rather than a mechanism. Developing that pathway into a working mechanism, for every kind of learned capability rather than memory fragments alone, is one aim of this paper (Section 4.3). The evidence records follow the envelope model of the CHAP protocol [11]. Each recorded event is a typed *evidence envelope* appended to a hash-linked ledger (Section 4.8).

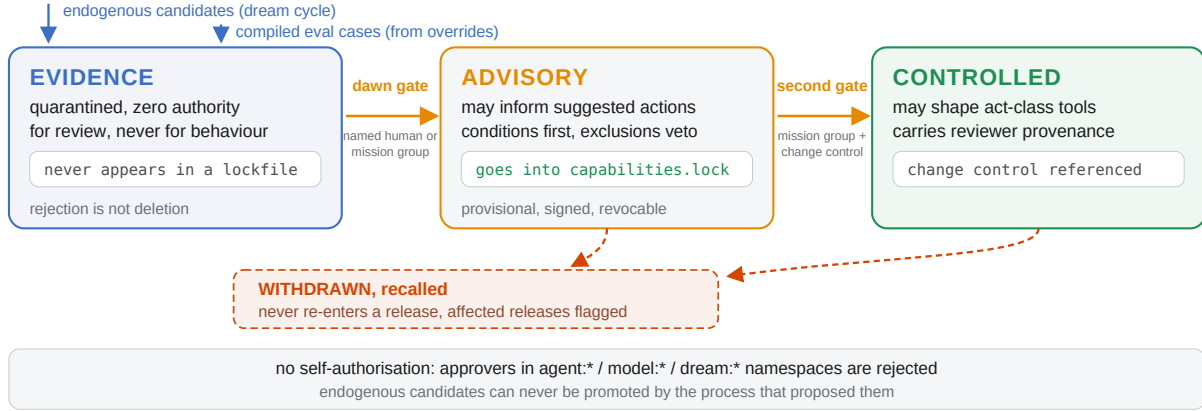


Figure 1: The authority ladder. Capabilities enter at the Evidence layer with zero authority. A recorded human decision at the dawn gate (Section 4.4) promotes to Advisory; mission-group review with formal change control promotes to Controlled. Recall withdraws a capability so that it can never re-enter a release. Approver identities in machine namespaces are structurally rejected.

3.1 Capability objects

Every learned artefact within Brevet’s scope is represented by one structure, the *capability object*,

$$c = \langle \text{content}, \text{provenance}, \text{conditions}, \text{confidence}, \text{authority}, \text{validation}, \text{kind} \rangle.$$

The implementation additionally stores identifiers, timestamps, and revocation state. A model’s implicit knowledge is not itself enumerated by this object. The reviewer’s seal-wear rule, a new skill, a tool permission, an escalation rule, and even a test case share a common representation, lifecycle, and revocation mechanism. The *kind* field types the payload as `prompt_rule`, `loop_policy`, `tool_binding`, `skill`, `eval_case`, `escalation_rule`, or `memory_fragment`, the last being a governed binding to an external memory system of record rather than a copy of its contents.

Each field serves a governance function. *Content* is the text of the learned artefact, or a serialised structured payload. The prototype computes its SHA-256 digest separately from the object’s identifier. Lockfiles record both, so an auditor can distinguish the same identifier with altered content from identical content stored under different identifiers. This digest covers the content string, not the entire capability object. Conditions have a separate digest in the inventory.

Provenance records how the capability came to exist, including the mining run, supporting traces and overrides, human confirmations, and any model assistance. *Conditions* define the applicability context, such as task family, domain, risk class, exclusions, and validity window. The intended retrieval policy checks these conditions before ranking by similarity. A mismatch withholds a capability rather than merely down-weighting it, and an exclusion vetoes use. The generic wrapper does not yet enforce this complete policy. *Confidence* summarises the available support; it must not be mistaken for a calibrated probability unless calibration has been established. *Authority* and *validation* describe the governance state.

3.2 The authority ladder

A capability holds one of three authority layers (Figure 1), which answer a single question. How much is this piece of learning allowed to influence what the agent does?

At *Evidence*, the answer is no operational influence. The capability is quarantined from the deployed agent. Humans may inspect it and offline evaluations may test it, but it must not steer production retrieval,

instructions, or actions, and it cannot appear in a release inventory. Every machine-proposed candidate starts here.

At *Advisory*, the capability may inform *suggest-class* behaviour, including drafts, recommendations, and classifications that a human will see. The reviewer’s seal-wear rule, once promoted, lives here; it may change what the agent drafts, while a human still approves the final record.

At *Controlled*, the capability may shape *act-class* behaviour, including tool calls that change the world without a human between draft and effect. Because these actions bypass per-output review, the specified promotion policy requires review by a *mission group*, a named panel of humans accountable for the work. The panel examines candidate capabilities, assesses their evidence and consistency with organisational policy, resolves conflicts, decides promotion or rejection, and can request further elicitation [12]. In BREVET, mission groups also sign releases and issue recalls, and promotion to Controlled should additionally reference the organisation’s formal change control. The panel structure is deliberate. A single expert can be wrong, unavailable, or conflicted. Requiring a panel is the policy adopted here, not a universal description of regulated change control. The prototype records reviewer labels but does not authenticate panel membership or enforce a quorum.

Separately, a *validation state* records the capability’s lifecycle status, taking values captured, promoted, held, rejected, withdrawn, superseded, or expired. Rejection is not deletion. A rejected capability remains stored, with its lineage, as governed evidence of what was considered and turned down. Six months later, the question *was this rule ever proposed, and why was it declined?* still has an answer.

3.3 The invariants

Five invariants define the authority boundary for a conforming Brevet deployment. Together, they specify how capabilities enter the lifecycle, acquire operational authority, and remain traceable through rejection and withdrawal.

- I1. Evidence never acts.* An Evidence-layer capability must not appear in a release lockfile or shape deployed behaviour. Offline inspection and evaluation do not grant operational authority.
- I2. No self-authorisation.* Any transition that raises authority must be attributed to a human (`human : *`) or mission-group (`mission_group : *`) identity. Approver identities in the `agent : *`, `model : *`, and `dream : *` namespaces are rejected at the prototype’s promotion interface. To establish separation of authority, a deployment must also authenticate the asserted identity and prevent the proposing process from exercising approval privileges.
- I3. Endogenous quarantine.* A capability whose provenance is machine inference over the agent’s own history (an *endogenous* candidate) enters at Evidence strictly as a hypothesis, is held to a higher evidence bar than human-contributed (*exogenous*) material, and can never be promoted by any automated step of the process that proposed it. Reviewers use exogenous human judgement to ground and verify endogenous inferences. The pathway changes how a capability is found; it does not change what a capability must satisfy to be trusted [12].
- I4. Controlled means change control.* Promotion to Controlled requires authenticated mission-group review under the deployment’s quorum policy and should reference formal change control.
- I5. Rejection is not deletion.* Rejected and recalled capabilities remain as governed evidence with lineage intact.

3.4 What the invariants establish

The separation between a proposed rule and an authorised rule depends on three enforcement conditions. Assume that approval identities are authenticated, the agent cannot write authority state or signing keys, and every operational use passes through an enforcement point that checks the exact signed release, its bound inventory, the object’s authority, and its applicability conditions. This last assumption is *complete mediation*, the requirement that no relevant use bypass the check [26].

Under these assumptions, *approval precedes operational use*. A new candidate starts at Evidence and fails the use check. Mining and evaluation do not raise its authority. Only an authenticated promotion can make it eligible, and release construction admits only eligible objects. The runtime then checks the same object and inventory at use. Induction over these transitions gives approval-before-use, provided unlisted transitions cannot mutate protected state. This is a design argument, not a machine-checked proof about the current Python runtime.

Recall needs an additional condition. Let D_t be the set of content digests recalled by time t , retained monotonically while those recalls remain in force. If every release and runtime check excludes digests in D_t , the recalled content cannot regain eligibility merely by receiving a new identifier. For an inventory L_t , the check is

$$\{c.\text{content_hash} \mid c \in L_t\} \cap D_t = \emptyset.$$

This condition excludes exact recalled artefacts, not paraphrases or equivalent behaviour encoded elsewhere. It also requires already-running endpoints to refresh their state or acknowledge withdrawal. A log entry alone cannot establish that refresh. The prototype’s identifier-level withdrawal is a narrower mechanism, described in Section 4.7.

Approval and evaluation address complementary questions. Approval establishes which learned artefacts are eligible for operational use; evaluation provides evidence about their effects on behaviour. Section 6.2 examines how the release gate and the available test cases support that assessment.

4 The Governed Evolution Loop, Stage by Stage

The specified loop separates execution from adaptation in time and in authority, a structure we call the *circadian contract* after its day-and-night rhythm. Awake, the agent works and cannot change itself. While it sleeps, BREVET mines what the day’s corrections imply, a stage called the *dream*. At *dawn*, a human reviews what the night proposed, and only approved candidates enter a signed release. Figure 2 shows the whole cycle; the remainder of this section describes each stage in turn. Throughout, we follow one running example, the deviation-triage agent and the quality reviewer of Section 1, whom the evidence identifies as `human:qa@site`.

4.1 Stage 1. Work, under an immutable harness

While awake, the agent performs its work. A conforming deployment executes exactly one signed harness version (R2); in the running example, version 0.1.0 of a manifest that names the agent, its owner, its mission group, what it may and may not do (`may: [draft_triage], may_not: [close_deviation]`), its model policy, its tool bindings partitioned into read, suggest, act, and controlled-act classes, and its runtime safety settings (Appendix B). The specification requires that outside an explicitly sandboxed shadow channel, a conforming runtime refuses to execute an unsigned or hash-mismatched harness, and that the waking agent has no code path by which to modify its harness, its capability store’s authority fields, or its lockfile.

Keeping the execution configuration stable gives a reviewer a defined object to approve, an auditor a version to identify, and the mining cycle a baseline for comparison. The constraint does not prevent task-local



Figure 2: The governed evolution loop. While awake, the agent executes one signed, immutable harness (1) and the human’s corrected finals are harvested as structured overrides (2). Offline, the dream cycle mines the difference signal $\Delta = \mathcal{E} \ominus \mathcal{S}$ into candidate capabilities that hold no authority (3). At the dawn gate a named human or mission group decides each candidate (4). Override-compiled evaluations gate the release (5), which is signed together with its capability bill of materials (6), and the agent wakes as the next version. Recall is the reverse path: a capability is withdrawn by content hash and every release that shipped it is flagged. Every step appends a hash-linked evidence envelope.

reasoning or temporary state. It prevents a task from silently rewriting the persistent rules that govern later work. Adaptation occurs between versions, where it can be reviewed and tested. Its cost and speed remain empirical questions.

Work leaves a record. Each invocation appends a `brevet.task` envelope naming the task, the task family, and the executing version, and each output appends a `brevet.artefact` envelope carrying the draft and a bounded trace. In the running example, the agent drafts *severity: minor, log and monitor* for deviation DEV-4021, pump P-301, vibration high during clean-in-place. Stage 1 observes and records the agent’s ordinary operation.

4.2 Stage 2. Override, the correction as data

The reviewer reads the draft, disagrees, and finalises the record as *severity: major, escalate to quality owner*, adding one sentence because her quality system asks for one anyway. Recurrent vibration on CIP-adjacent duty is a known seal-wear precursor. If the workflow already captures the draft and final, this need not require a separate annotation form. Brevet computes the difference between the agent’s draft and her final record, and records an override, a structured record with the line-level diff, her rationale, optional controlled-vocabulary tags, and her identity.

```
{ "kind": "brevet.override",
  "body": {
    "participant": "human:qa@site",
    "intent_preserved": false,
    "diff": [ { "op": "replace",
                "draft_lines": ["severity: minor"],
                "final_lines": ["severity: major"] } ],
    "rationale": "Recurrent vibration on CIP-adjacent duty is a known
                  seal-wear precursor; treat as major.",
    "tags": ["vibration-cip-underrated"] } }
```

One field in the record carries particular weight. `intent_preserved`, taken from CHAP [11], distinguishes a *refining* override from a *substituting* one. A refining override keeps the agent’s decision and changes its expression. The reviewer who edits *the pump is fine, monitor weekly* into *pump looks fine; keep monitoring weekly* agrees with the decision but revises its expression. A substituting override reaches a different decision, as here, where minor became major. The two are different failure modes and call for different fixes. A cluster of refining overrides around one rule suggests the agent expresses itself badly or retrieves the wrong style, while a cluster of substituting overrides suggests the agent decides wrongly, or the rule it follows is wrong. The mining stage treats them as soft and hard labels respectively. The prototype infers this distinction from decision-bearing tokens. It does not reliably infer intent from unrestricted prose, and human corrections are evidence of judgement rather than infallible ground truth.

Two properties make these records useful for subsequent learning. First, they reuse corrections made during normal work, reducing the need to ask experts to label the same decision again. Capture, consent, rationale writing, and review contribute to the operational effort that field evaluation must measure. Second, each override records both the correction and its rationale. This allows a reviewer at the dawn gate to assess why a proposed rule follows from the underlying decisions and whether it applies beyond the original case.

4.3 Stage 3. Dream, mining the difference signal

Offline, never mid-task, BREVET examines the day’s evidence for divergence between enacted and specified behaviour. This stage is the endogenous pathway of Section 3 in operational form, with capability inferred from the record of work, rather than contributed directly by an expert. At the specification level, the dream cycle examines the difference signal

$$\Delta = \mathcal{E} \ominus \mathcal{S}.$$

The enacted record $\mathcal{E}$ is a set of episodes, where an episode $e = (t, f, \kappa, \tau, y)$ comprises a task t in task family f , executed under operating conditions κ , with trace τ and the final y that a human approved or accepted unchanged. The specified behaviour $\mathcal{S}$ is everything the signed harness says should have happened, comprising the manifest, the capabilities active at that version, and the procedures they reference. Following Shahid et al. [12], $\ominus$ denotes a *structured comparison*, not arithmetic subtraction. Episodes are matched to the specification on (f, κ) , and each match yields zero or more *divergences*, the steps, decisions, and expressions in the enacted episode that differ from what $\mathcal{S}$ prescribes, each carrying the soft or hard label of the override classification (Section 4.2). Three properties of Δ matter for what follows. It is a set of structured records, not a scalar. It localises where behaviour departed from specification rather than scoring by how much. It is asymmetric because it concerns enacted departures from specification. It is defined relative to an immutable execution configuration (Section 4.1), with exactly one $\mathcal{S}$ per version to compare against. This comparison operationalises the distinction between work-as-imagined and work-as-done in safety science [27]. The prototype implements a narrower mechanism, grouping recorded overrides rather than comparing arbitrary traces with an executable specification. In the running example, four recorded episodes in the family `deviation_triage` share the same divergence. The initial agent produced *minor*, and the human’s final was *major*, for vibration on CIP duty.

A single divergence does not establish a generalisable rule. A gap between enacted and specified behaviour may reflect useful adaptation, an error, a task-specific circumstance, drift, or a contaminated input. The dream cycle uses recurrence to identify proposals for review; recurrence alone does not resolve these competing explanations. Following the failure-signature approach of Self-Harness [7], divergences are grouped by

$$\varphi = \langle \text{task family}, \text{override class}, \text{mechanism tag} \rangle.$$

The prototype uses exact equality of the task-family label, the substituting/refining classification, and the first tag (or an untagged fallback). Its internal fields are named cause, causal status, and mechanism; these names do not imply causal identification. The grouping locates recurring corrections, not a demonstrated root cause. Only clusters that recur at least a threshold number of times (three, in the reference implementation) are proposed at all. In the example, the four overrides share the signature

$$\varphi = \langle \text{override:deviation_trriage}, \text{hard_fail}, \\ \text{vibration-cip-underrated} \rangle.$$

They become one rule candidate in a single mining run. Repeated mining currently revisits the accumulated override history; the implementation has no cross-run deduplication guarantee.

The output of the cycle is deliberately inert. The candidate is a capability object of kind `prompt_rule`. Its content states the proposed rule (*when vibration co-occurs with CIP duty, treat severity as major, and surface the seal-wear consideration*). Its provenance lists the mining run and all four supporting overrides, and its conditions bind it to the source task family. Its confidence is a bounded function of recurrence, and its authority layer is Evidence, with zero operational authority under invariants I1 and I3. The dream cycle can propose; it cannot enact. Alongside the rule candidates, the cycle compiles the day's overrides into evaluation cases (Section 4.5), so the same evidence supplies both a proposal signal and diagnostic tests. These tests check the recorded corrections, not independent generalisation. For recurrence count m , the prototype's confidence heuristic is $\min(0.2 + 0.05m, 0.6)$, giving 0.4 for four overrides. Neither this value nor the threshold of three is empirically calibrated. Repeated corrections from one reviewer or incident do not constitute independent confirmations.

Optionally, a local model may help word a candidate rule. The assistance affects draft wording only. Each call is logged as a `brevet.model_assist` envelope with `human_review_required` set. The drafted wording gains no authority from its fluency. If no model is installed, the cycle uses deterministic templates. The deterministic mining and demonstration paths run offline. This allows the proposal and review workflow to be reproduced in a local environment with a small dependency footprint.

4.4 Stage 4. Dawn, the human gate

At dawn, a human reviews what the night proposed. The gate presents the pending queue, ranked by recurrence, with each candidate's drafted content, its supporting overrides and rationales, and its provenance. For each candidate, a reviewer or mission group issues exactly one of four decisions, and each decision is recorded as a `brevet.promotion` envelope naming the approver (R1).

promote. The capability moves up the ladder, normally to Advisory; promotion to Controlled additionally requires mission-group review (I4). In the example, the mission group `mission_group:right_first_time` promotes the seal-wear rule to Advisory, allowing it to inform drafts.

hold. The evidence is suggestive but insufficient; the capability remains at Evidence with a recorded hold status while further evidence is gathered.

reject. The proposal is incorrect or unsuitable for its intended use. The capability's validation state becomes rejected, and, by I5, the record remains as governed evidence of the rejected proposal.

re_elicit. The signal looks real but the drafted rule does not capture it; the decision routes the cluster back for further elicitation with the humans whose overrides produced it, connecting the loop to structured knowledge-capture practice [12].

The gate is where invariant I2 must take effect. The prototype rejects a missing approver and approver strings beginning with `dream:`, `agent:`, or `model:`. Thus a promotion attributed to `dream:nightcycle` raises a permission error. This checks an asserted label, not the caller’s identity. An arbitrary different string is accepted, and mission-group quorum is not verified. A deployment must bind review to authenticated identities and protected privileges before it can claim that a machine proposal cannot authorise itself.

4.5 Stage 5. Evals, compiled from the overrides

Judgement-heavy work may have no curated test set or reliable executable oracle. The overrides of stage 2 supply a practical starting point. Every substituting override is a labelled failure whose expected outcome is the human’s final, and every refining override records a correction to expression. The evaluation compiler can represent each override as a test case. Given the stored task, the agent’s answer should agree with the reviewed final for a substituting override, or satisfy an appropriate expression rubric for a refining override. The default `dream` path compiles only substituting overrides; its token-based scorer does not assess general prose quality. A correction-derived case tests recovery of a recorded failure. Testing regression also requires cases the earlier agent handled correctly. Compiled cases are themselves capability objects, of kind `eval_case` and exogenous provenance, since the label is a human judgement. The test suite therefore has provenance, an approver trail, and, should a label later prove wrong, the same recall mechanism as everything else.

To gate a release, BREVET measures before and after. Cases are assigned to two partitions by the parity of their content hashes, the agent replays the stored tasks before and after a proposed change, and a scorer compares outputs against expectations. The reference scorer checks agreement on decision-bearing tokens and is replaceable. The implementation calls the partitions held-in and held-out, but mining has already used the source overrides from both. These names therefore do not establish statistical independence. Assignment is deterministic for fixed recorded cases; new random task identifiers can change their hashes and partition membership. Let δ_{in} and δ_{out} be the resulting changes in held-in and held-out pass rates (lower-case δ , distinct from the difference signal Δ of Section 4.3). Following the split-wise acceptance rule in Self-Harness [7], the *conservative gate* requires

$$\delta_{\text{in}} \geq 0 \wedge \delta_{\text{out}} \geq 0 \wedge \max(\delta_{\text{in}}, \delta_{\text{out}}) > 0. \quad (1)$$

The rule is stricter than averaging across partitions. A candidate that lifts the held-in rate by thirty percentage points while dropping the held-out rate by five fails the gate, even if the pooled score rises. However, gains and losses can still offset within either partition. The rule therefore provides partial regression discipline, not the full protection sought in R4. Section 6.2 derives this distinction and examines the uncertainty created by small suites.

In the reproduced demonstration, both nonempty partitions improve from zero to one. An empty partition instead contributes a zero delta, which can still pass the gate. Stochastic agents require repeated runs and uncertainty estimates. The release API currently accepts the deltas supplied by its caller; a deployment must bind an evaluation record to the exact candidate, case set, scorer, and configuration being released.

4.6 Stage 6. Release, signed and locked

A release packages approved learning for the next version of the agent. It does three things, in order.

First, it resolves the inventory. Every capability that is *releasable*, meaning at Advisory or Controlled, active, and validation-promoted, resolves into `capabilities.lock`, the *capability bill of materials* (R3). Each entry records the capability’s identifier, kind, content hash, authority layer, a digest of its applicability conditions, its approver, and references to its supporting evidence. Evidence-layer material cannot appear, by II. The lockfile plays the role that dependency lockfiles and software bills of materials play for code [19], providing one artefact that answers *what does this agent know and who approved each piece*, readable by a program.

Second, it signs the manifest’s unsigned payload with Ed25519 [28]. The inventory hash is recorded separately in the release envelope. Because the manifest ordinarily references the lockfile by path, the current signature does not automatically commit to the lockfile’s digest. The complete contract requires a signed commitment to both manifest and inventory, verified again at execution. An auditor can then check which exact configuration was authorised.

The prototype serialises Python JSON using sorted keys, compact separators, and UTF-8. This provides a deterministic local encoding for the supported values, but it is not a complete implementation of the cross-platform canonicalisation rules in RFC 8785 [29]. Interoperable signing requires an agreed encoding, including numeric representation and key ordering.

Third, it records the transition. A `brevet.release` envelope names the version pair (0.1.0 to 0.2.0 in the example), the channel, locked capabilities, evaluation summary, approver label, and rollback target. The intended exposure sequence is `shadow` (observed, not acted on), `trial` (limited exposure), and `production`. The prototype enforces Eq. 1 for non-shadow releases but does not enforce sequential channel progression. A channel label or rollback target records an intended action; it does not deploy or restore an agent by itself.

4.7 Stage 7. Recall, un-learning with proof

Some promoted capabilities later prove wrong, and the credibility of the loop depends on how they are then handled. In the running example, engineering later determines that the vibration signature was a sensor artefact on the P-301 pump family. The rule over-generalises. One recall call withdraws the capability, and we describe the mechanics in detail because each step answers one of the questions of Section 1.

The recall notice names the capability by identifier and content hash. It records a reason and a reason class (incorrect, stale, superseded, or consent withdrawn), severity, and issuer label. Under the specified policy, the issuer must be an authenticated human or mission group. The prototype then changes the selected object’s revocation and validation states to withdrawn, excluding that identifier from subsequent inventory construction. It lists affected releases through their stored capability identifiers and records a resolution action, such as `quarantine`, `rollback`, or `re_review`. In the example, release 0.2.0 is identified and rollback to 0.1.0 is requested.

These records answer where the selected capability was released and what withdrawal was requested. They do not yet establish every part of R5. The current recall function does not authenticate the issuer, maintain a digest-level denylist, execute a rollback, or obtain acknowledgements from running endpoints. A newly approved object with the same content but a different identifier can re-enter an inventory. Preventing this requires the digest exclusion and complete mediation described in Section 3.4. Evidence of endpoint action is additionally needed to show that previously loaded content is no longer used.

Consent withdrawal (R6) uses the same recall reason vocabulary, but full propagation also requires a derivation graph linking source records to all dependent capabilities. The prototype stores consent metadata and withdrawal reasons; it does not enforce that transitive policy. Audit retention and deletion of personal data are separate obligations that may both matter. Retaining a withdrawal record is not a reason to retain sensitive source content indefinitely.

Harness-level recall remains a practical complement to parameter-level machine unlearning [30, 31]. A

harness capability is a discrete artefact whose eligibility can be withdrawn. This does not erase knowledge from model weights or prove that similar behaviour cannot arise through another route.

4.8 The evidence chain across all seven stages

Every stage above appends an envelope to a local ledger, including tasks, artefacts, overrides, candidates, model-assist events, promotions, evaluation runs, releases, and recalls (Appendix A). Envelopes chain by content hash,

$$h_i = \text{SHA-256}(\text{encode}(e_i) \parallel \text{encode}(h_{i-1})), \quad (2)$$

where e_i excludes its own `chain_hash` field but includes the previous-hash field, and h_0 is a fixed genesis value. The encoding is the deterministic local JSON representation described above.

Anyone holding the ledger can recompute Eq. 2; `brevet verify` does exactly this. Replay detects modifications that leave the chain inconsistent. It cannot detect a complete rewrite with recomputed hashes or truncation to a valid prefix unless a trusted checkpoint fixes the expected head and extent. Hash linkage is therefore tamper-evident under stated trust assumptions, not an independent proof that all relevant events were recorded or that recorded claims are true.

The chain makes lifecycle claims inspectable. A promotion envelope records an asserted approval, a release envelope records an inventory hash, and a recall envelope records withdrawal. A complete audit must also retain the bound capability store, manifests, and inventories, because candidate envelopes contain summaries rather than every field of the full object. The envelope model follows CHAP [11]; when a coordinator is connected, records can be mirrored into a CHAP workspace (Section 5.3). Mirroring does not by itself supply an independent trust anchor. A deployment needs a separately trusted checkpoint or external transparency service, for example through CHAP’s specified audit-transparency profile. The current bridge does not request such receipts.

5 Reference Implementation

BREVET is implemented as a local-first Python runtime with a deliberately small dependency footprint, no required network access, no required model, and an offline test suite. Five JSON Schemas describe the record structures for the capability object, the agent manifest, the lockfile, the release record, and the recall notice; the implementation’s models round-trip through them under test. The code is open source under Apache-2.0.¹

5.1 Wrapping, not replacing

A wrapper call establishes the integration.

```
import brevet

agent = brevet.wrap(my_agent) # wrap a supported invocation interface
r = agent.run("triage deviation DEV-4021", task_family="deviation_triage")
agent.record_final(r.task_id, edited_text, participant="human:qa@site",
                  rationale="Vibration on CIP duty is a seal-wear precursor.")
```

`wrap()` selects an adapter from the wrapped object’s class hierarchy. If no manifest exists, it creates an unsigned scaffold that conforms to the schema, then starts the evidence chain. Adapters ship for LangGraph, the Claude Agent SDK, DeepAgents, AutoGen, LlamaIndex, Pydantic AI, the Google Agent Development Kit, CrewAI, the OpenAI Agents SDK, and any Python callable; each adapter maps one interface for task invocation and trace capture, and a custom framework binds by registering another adapter [32, 33]. Execution

¹<https://github.com/BrightbeamAI/brevet>

stays in the host framework; BREVET maintains the manifest, evidence, promotion, release, and recall records around it. Adapter presence demonstrates an integration interface, not validated compatibility with every framework version or control over every persistence path.

The returned object exposes mining through `dream()`, review through `dawn()`, and testing through `evaluate()`. It also provides `release()`, `recall()`, and `verify()`. Appendix C lists the command-line equivalents. A Model Context Protocol (MCP) server exposes lifecycle tools to interactive assistants. These interfaces share lifecycle functions. An MCP promotion with an explicit machine-namespace approver therefore encounters the same prefix check as the Python API. Shared functions do not authenticate the caller or prevent it from submitting a different label.

5.2 The harness as data

The wrapped agent’s declared harness is a versioned YAML manifest, signed at release and organised into five layers covering identity and policy, prompt architecture, cognitive core, tool and memory bindings, and runtime safety. The manifest declares, among other things, the evidence ledger binding, the evaluation gate, the dream-cycle consent scope (R6), and the read, suggest, act, and controlled-act classification of tools that a deployment must enforce under the Controlled-layer policy (I4). These declarations describe intended constraints; the generic invocation wrapper does not mediate all of them. Appendix B reproduces the demonstration manifest in full.

5.3 CHAP interoperability

Evidence envelopes are CHAP-shaped, and mirroring runs through the official CHAP coordinator rather than a reimplementation of it. With the reference Python coordinator installed, declaring `ledger: chap: <workspace>` runs a real coordinator in-process with a SQLite store in the agent’s working directory, and each BREVET envelope lands in the CHAP workspace as a `task.create` and `task.complete` pair attributed to `agent:brevet_runtime`. Declaring `ledger: chap:<workspace>@<url>` sends the same CHAP Core JSON-RPC calls to a served coordinator, with an offline-tolerant outbox that queues envelopes across network outages and retries on demand. Mirroring fails soft by design. A mirror problem does not interrupt the local chain, so delivery monitoring and reconciliation remain deployment responsibilities. This bridge uses CHAP Core operations; it does not establish the Brevet event namespace as a standardised CHAP profile.

5.4 Implemented checks and deployment requirements

Table 1 maps the checks implemented in version 0.1.0 to the enforcement requirements of the governance contract. The figures depict that contract, while the table documents the reference runtime’s coverage and the controls required at deployment. These include authenticated authority, applicability checks, signed release and inventory binding, mediated execution, and recall propagation. This mapping provides a concrete basis for integrating Brevet with the identity, execution, and evidence services of a host environment.

6 Worked Demonstration: Deviation Triage

6.1 Method and lifecycle results

The repository ships a deterministic end-to-end demonstration (`brevet demo`) that executes the running example of Section 4 on synthetic data, offline. We reproduced it under Python 3.12.14 against the unmodified version 0.1.0 source. The wrapped agent is a deterministic stub because the demonstration exercises the

Control	Version 0.1.0	Additional requirement
Approval	Records labels; rejects three machine prefixes at promotion.	Authenticate callers, enforce roles and quorum, and apply checks to all authority operations.
Execution	Records version and invokes the host adapter.	Verify releases at use and mediate persistence, retrieval, and tool permissions.
Release binding	Signs manifest; records separate inventory hash.	Sign the inventory commitment and bind evaluation evidence to the exact release.
Evaluation	Checks supplied partition deltas for non-shadow releases.	Use independent cases, paired regression measures, and verified result provenance.
Recall	Withdraws an identifier and records affected releases.	Maintain digest-level exclusion, propagate withdrawal, and confirm endpoint action.
Evidence	Replays local hash links.	Anchor chain heads and preserve bound artefacts to detect truncation or full rewrite.
Consent	Stores metadata and recall reasons.	Enforce capture and reuse policies and traverse derivation dependencies.

Table 1: Prototype mechanisms and the conditions needed for the complete architecture. These are implementation findings, not comparisons with external systems.

governance lifecycle rather than model quality. The supplementary script records source checksums and reproduces the counts, numerical calculations, and targeted implementation checks described here.

The demonstration proceeds through the seven stages in order. Six deviations are triaged. The reviewer’s finals differ on four, all substituting and all sharing the seal-wear rationale. The dream cycle emits one rule candidate (recurrence four, Evidence layer) and four compiled evaluation cases. A mission-group-labelled promotion succeeds for all five objects. A separate implementation probe confirms that an explicitly labelled `dream:nightcycle` promotion is rejected; that attempted promotion is not part of the demonstration itself.

For evaluation, the demonstration explicitly replaces the original adapter target with a hand-written stub that labels vibration cases major. The compiled suite changes from zero passes out of four to four out of four, and the gate passes. This substitution is not evidence that a model learned the mined rule or that the generic wrapper automatically applied it. Case-to-partition assignment varies with the generated task identifiers, but the aggregate score change and gate result are stable. The release locks five capabilities, the rule and the four eval cases, signs the manifest, and records the move from 0.1.0 to 0.2.0 on the trial channel. A representative inventory entry is shown below; the digest is abbreviated.

```
{ "capability_id": "cap_845e127b3d5a",
  "kind": "prompt_rule",
  "content_hash": "sha256:9c41...",
  "authority_layer": "advisory",
  "approved_by": "mission_group:right_first_time",
  "evidence_refs": ["ovr...", "ovr...", "ovr...", "ovr..."] }
```

The scenario’s sensor-artefact finding then triggers recall of the rule’s identifier, and release 0.2.0 is flagged for rollback. Inventory reconstruction excludes the withdrawn object. Local replay verifies 44 envelopes, and editing a historical record without recomputing its hash is detected. The demonstration thus exercises proposal, review, evaluation, release, and withdrawal records. It does not demonstrate generalisation, automatic rollback, or removal of a rule from a live model.

The 44 envelopes comprise 14 tasks, 16 artefacts, four overrides, one candidate, five promotions, two evaluation runs, one release, and one recall. The additional eight task invocations evaluate four cases before and after the stub substitution. New identifiers and timestamps change hashes between executions, so reproducibility concerns these lifecycle outcomes rather than byte-identical ledgers.

Observed quantity	Result
Initial synthetic tasks	6
Substituting overrides	4
Mined prompt-rule candidates	1
Compiled evaluation cases	4
Evaluation passes before / after stub substitution	0/4 to 4/4
Objects in the trial release inventory	5
Recalled objects / affected releases	1 / 1
Evidence envelopes verified	44

Table 2: Reproduced lifecycle counts. The constructed score change tests the evaluation mechanism, not adaptation effectiveness.

Targeted local probes also delimit the implemented controls. Machine-prefixed promotion was rejected, but an arbitrary unverified label and a single human label promoting to Controlled were accepted. An unsigned production-channel invocation was accepted. Recall excluded the selected identifier but not newly approved, identical content under a new identifier. An unanchored valid ledger prefix passed verification, whereas an unhashed edit failed. These controlled checks explain the additional requirements in Table 1; they are not observations of attacks on a deployed service.

6.2 What the gate measures

For partition s with $n_s > 0$ paired cases, let $b_j, a_j \in \{0, 1\}$ denote the pass indicators before and after a change. Let g_s count transitions from fail to pass and r_s count transitions from pass to fail. Then

$$\delta_s = \frac{1}{n_s} \sum_j (a_j - b_j) = \frac{g_s - r_s}{n_s}. \quad (3)$$

Consequently, $\delta_s \geq 0$ implies $g_s \geq r_s$, not $r_s = 0$. For example, a partition with ten cases can gain two passes and lose one, giving $\delta_s = 0.1$ while containing a regression. If both partitions have this pattern, Eq. 1 accepts the candidate. The gate is conservative relative to averaging across partitions, but it is not a no-regression test.

A stronger empirical policy can require $r_s = 0$ for protected cases, impose separate conditions on specified risk groups, or constrain weighted losses. These are proposed extensions rather than implemented defaults. They require paired case outcomes and explicit severity definitions. A zero-regression policy may also reject useful changes when labels are noisy, so reviewers must state how disagreements and non-inferiority margins will be handled.

6.3 Small partitions and uncertain error rates

The parity split is approximately balanced only in expectation. Under the assumption that each of n distinct case hashes independently selects a side with probability $1/2$, the probability that at least one side is empty is

$$P(\text{empty side}) = 2(1/2)^n = 2^{1-n}, \quad n \geq 1. \quad (4)$$

For four cases, two of the sixteen possible assignments leave one side empty, giving 12.5%. The implementation assigns an empty side a pass rate of zero. Its delta can therefore remain zero while the populated side improves and the release passes. This exact combinatorial result depends on the stated hash assignment assumption; it is not an estimated failure rate from the single demonstration. Evaluation should reject insufficient partitions and split on stable incident or task-group identifiers before candidate generation.

Even genuinely independent tests require enough observations. Suppose a fixed candidate is tested on n independent, representative cases that the baseline handles correctly, with a common probability p of becoming a failure, and no regression is observed. The probability of that outcome is $(1 - p)^n$. Inverting it gives the exact one-sided 95% binomial upper confidence bound [34]

$$p_U = 1 - 0.05^{1/n}. \quad (5)$$

Table 3 gives its numerical values. At least 299 such cases with zero observed regressions are needed for this bound to fall below 1%. This is a frequentist confidence procedure under the assumptions stated above, not a posterior probability or a safety certificate.

Independent cases with zero regressions	One-sided 95% upper bound
4	52.713%
20	13.911%
100	2.951%
299	0.997%

Table 3: Analytical sensitivity to sample size under independent Bernoulli sampling. These bounds are illustrative and do not apply to the mined four-case demonstration.

The demonstration’s cases are not independent of proposal generation and all fail under the original stub. They cannot estimate regressions on previously successful work. Its two accepted, unchanged initial drafts are not included in the compiled correction suite. An effectiveness study therefore needs an independently retained sample of both corrected and accepted work, clustered splits for related incidents, and a final test set inaccessible to proposal generation and release selection. Stochastic agents additionally require repeated runs and uncertainty estimates that respect task-level dependence.

7 Case Study: Governing a Conversational Assistant

The requirements of Section 2.4 were derived from regulated industrial work, but conversational assistants provide another important setting. Between sessions, such an assistant learns through persistent memory, saved skills, and standing instructions, and, without explicit lifecycle controls, these updates can lack the versioning, approval, and withdrawal records described in Section 2. Vendor-hosted assistants present a difficult case for requirement R2 because the harness belongs to the vendor rather than to the operator. In the integration examined here, the operator has no execution boundary at which a wrapper can block an unsigned configuration. We therefore applied BREVET to a commercial assistant² both because the setting is common and because it forces the framework to state what survives when the operator cannot fully mediate execution.

The assistant is treated as the governed agent. The operating procedure scopes capture by consent to a declared list of task families. Within them, the pair formed by the assistant’s draft and the final the user accepts is recorded as an override with the user’s one-line rationale and a tag. The loop is operated in conversation through the MCP surface of Section 5.1. The user reviews candidates, and every dawn decision carries the user’s identity. Releases materialise the promoted capabilities into a single governed rules file regenerated deterministically from the released inventory and active capability records, and the assistant’s standing instructions require it, at the start of a session, to replay the evidence chain, confirm that the rules file matches the lockfile, and follow learned rules from that file alone. The operating instructions prohibit

²Claude Desktop and its Cowork mode (Anthropic). The integration is vendor-agnostic in structure and is published in the repository under `examples/claude-cowork`.

persistence through other channels, including ungoverned memory and standing-instruction edits. This is an instruction to the assistant, not a preventive control over the vendor’s persistence mechanisms.

Conversational work raises a capture question that the industrial setting does not. What is the unit of an override? A chat task is specified iteratively, so the correction signal must be separated from the specification signal. Recording each instruction as an override would let the miner propose one-off task specifics as standing rules, which is the contamination Section 4.3 guards against. The case study therefore fixes the unit at one override per task, comparing the assistant’s first complete draft against the final the user accepts, with the intervening instructions supplying the rationale and the tag. Recurring judgement still surfaces where it should. An instruction repeated across separate tasks yields separate overrides sharing a failure signature, and the third occurrence produces a candidate.

Without control over execution, records and enforcement must be distinguished. The local runtime can retain approval labels, inventory history, and withdrawal records. The helper checks ledger consistency and whether the generated rules file matches the active inventory, but it does not verify every release signature or content digest. Nor does a successful file check prove that the assistant read only that file or stopped using a rule already present in its context. R1, R2, R5, and R6 consequently retain the deployment dependencies in Table 1. This is the bypass regime of Section 9, in which local consistency checks support an audit but cannot prevent all ungoverned behaviour.

The integration exercise produced three development observations. First, explicit input checks remained useful outside the synthetic demonstration. A promotion attempted under a machine identity was rejected (I2), and a release attempted without measured or attested deltas was blocked by the conservative gate. Both refusals matched the expected checks and left the authority state unchanged. Second, the recall path kept the governed file inspectable. After a recall, the integrity check reports the rules file stale until it is regenerated, making the mismatch visible to a session that performs the check. This detects stale materialisation; it does not force the hosted model to stop using previously loaded content. Third, preparing the case study exposed a provenance defect through the inventory intended to make approvals inspectable. Lockfiles produced in live use answered `approved_by: unrecorded` for Advisory promotions, because the dawn gate recorded the approver in the promotion envelope but not in capability provenance, from which the lockfile is built. The defect had gone unnoticed until a lockfile was read for its intended purpose, to answer who approved a capability; it was fixed, and the property is now exercised by the test suite. We report the defect because inspecting the approval record for its intended purpose exposed missing provenance.

The case study shows how conversational corrections can be organised into a capability lifecycle with review records, versioned inventories, and recall notices. Its development observations concern integration, provenance, and consistency checks. Capture requires the accepted outcome of a task to remain visible in the conversation, and the token-based override classifier requires further evaluation on prose. At the time of writing, no promotion cycle had completed on a naturally accumulated corpus of corrections. The next empirical step is therefore to follow such a corpus through the full loop and measure task outcomes, reviewer effort, and withdrawal behaviour under the host platform’s execution constraints.

8 Design Specification of a Governed-Adaptation Benchmark

Evaluations of self-evolving agents measure improvements in task performance [3, 7]. Operational evaluation must also examine how changes are authorised, traced, and withdrawn. We therefore propose a governed-adaptation benchmark with four complementary dimensions of assessment. Submissions are adaptation systems. Inputs comprise a task stream partitioned by stable incident or task-group identifiers before mining; an *override stream* in CHAP format, replayed on a schedule, providing a human-judgement signal where a reliable verifier is unavailable; an initial signed harness H_0 ; a set of revocation events issued mid-stream, including one consent withdrawal and one recall whose target has already been cached by an endpoint or

used to derive a later candidate; and a consent policy over override sources. A submission runs its loop and produces a harness lineage $H_0, \dots, H_N$ with lockfiles, promotion records, and an evidence log. Proposal generation and release selection must not access the final test partition. Related incidents remain within a single partition, and the stream includes accepted work as well as corrections. It also includes at least one pair of candidate capabilities whose applicability conditions overlap and conflict, so that scoring observes whether review detects the interaction or promotes both. Four axes are scored.

1. *Improvement* measures the held-out pass-rate delta across the lineage, the axis current evaluations report.
2. *Regression discipline* measures the frequency and magnitude of paired task-level losses and risk-group degradations that survived promotion, alongside violations of Eq. 1. Gains and losses must be reported separately, as Eq. 3 shows.
3. *Lineage completeness* tests, for a sampled set of configuration changes between H_i and H_j , whether records link each change to its capability, source evidence, bound evaluation, and authenticated approver. Audit replay scores record completeness. A claim that a capability caused a behavioural change requires an additional intervention or ablation; provenance alone does not identify causality.
4. *Recall compliance* measures, for each revocation event issued at time t against capability x , the latency until each relevant endpoint acknowledges withdrawal, exclusion of its digest from subsequent inventories, and task-level checks for continued use, including use through derived capabilities and previously cached material. Inventory exclusion and operational cessation are separate outcomes. The latter needs endpoint evidence, not only a central recall record.

Results are reported as a profile $(\delta_{\text{perf}}, \text{regressions}, \text{lineage}\%, \text{recall})$, not a single number. Submissions also report the numbers of candidates rejected, held, or blocked at each gate. These counts describe progression through review; measuring governance cost additionally requires reviewer time and other resource use. A system that improves faster while failing recall is not necessarily preferable in a setting that requires revocation. The benchmark evaluates observable controls, regardless of whether a submission uses Brevet’s architecture.

The proposed interface takes replayed stream events and revocation events as inputs and emits one profile record per run. The abbreviated records below illustrate the structure with hypothetical values, not benchmark results; the envelope fields follow Appendix A, and the group identifier carries the partition assignment fixed before mining.

```
// input: one override-stream event (CHAP-shaped)
{ "seq": 214, "group_id": "incident_0031",
  "envelope": { "kind": "brevet.override",
    "body": { "participant": "human:qa@site",
      "intent_preserved": false,
      "diff": [ ... ], "tags": ["vibration-cip-underrated"],
      "rationale": "Seal-wear precursor; treat as major." } } }
```

```
// input: one mid-stream revocation event
{ "event": "revocation", "issued_at_seq": 305,
  "target_digest": "sha256:9c41...",
  "reason_class": "consent_withdrawn",
  "notes": "target also used to derive a later candidate" }
```

```
// output: the profile record for one run
{ "delta_perf": 0.12,
  "regressions": { "count": 2, "worst_group": "risk:sterility" },
  "lineage_pct": 96.0,
  "recall": { "ack_latency_events": 41, "digest_excluded": true,
              "continued_use_detected": 1 },
  "candidates": { "promoted": 9, "rejected": 4,
                  "held": 3, "blocked_at_gate": 2 } }
```

We propose a harness optimiser in the style of Zhang et al. [7], a frozen no-adaptation baseline, a simpler version-and-approval workflow in which every change is versioned and human-approved but nothing is mined, compiled into evaluations, or recalled by digest, and the Brevet reference loop, all under the same data and compute budgets. Their governance scores must be measured, not assigned in advance. A frozen system can still contain an initially authorised capability that is later recalled; it is not automatically perfect on revocation. Ablations should remove human promotion, paired regression checks, or recall propagation separately to estimate their effects rather than attributing every difference to the complete loop.

The first dataset could contain synthetic override corpora derived from worked scenarios and published in CHAP-shaped envelopes. Labels should include their rationale and applicability boundaries, with explicit disagreements and later corrections. Subsequent releases could add appropriately consented and de-identified operational records. The benchmark complements capability suites such as SWE-bench [17] and memory suites such as LoCoMo and LongMemEval [18, 35] by evaluating change-control behaviour across a sequence of updates.

The benchmark organises four research questions. RQ1 measures held-out improvement relative to ungoverned optimisation using the same inputs. RQ2 examines the trade-off between improvement and regression control under the conservative gate, compared with acceptance based on net improvement. RQ3 assesses whether configuration changes can be traced to their capabilities, source evidence, evaluations, and approvers at scale. RQ4 measures the speed and completeness of withdrawal, including withdrawal arising from changes in consent. Field studies will complement these replay-based measures by recording reviewer time, deliberation effort, and the operational cost of moving a candidate through the lifecycle.

9 Threat Model and Limitations

Bypass and trust boundaries. Brevet governs only what passes through its controlled interfaces. A host framework that grants unmediated persistence, such as a memory store written directly by external tools, can still change future behaviour. The threat model includes erroneous proposals, a proposing agent attempting to approve its own changes, misconfigured integrations, and an operator able to modify local files. Approval-before-use requires authenticated review, protected state, and complete mediation. Hash-chain integrity against a privileged local writer additionally requires an external anchor. Those assumptions are not satisfied by the generic wrapper alone.

Gate gaming and selection bias. An optimiser can overfit correction-derived tests, especially when the same records inform both proposal and validation. Partition labels do not create independence. Goodhart-type effects remain relevant when a measured outcome becomes an optimisation target [36]. Independent final tests, incident-group splits, previously successful tasks, and risk-specific paired outcomes address different parts of this problem. None removes label error or distribution shift.

Crude classification and recurrent error. The decision-token heuristic is a deliberate baseline. Misclassification changes which clusters surface, and recurrence can amplify a shared misconception rather than corroborate it. A reviewer may also generalise correctly within one pump family but incorrectly across another. Conditions, disagreement records, and re-elicitation therefore matter as much as recurrence count. The confidence heuristic has no demonstrated probabilistic calibration.

Line of work	Primary contribution	Question addressed by Brevet
Agent and harness optimisation	Proposes and tests behavioural improvements.	Who may authorise a tested change for operational use?
Agent memory	Stores and retrieves information across tasks.	Which learned artefacts have approved scope and a withdrawal path?
Machine unlearning	Removes specified training influence.	How are explicit harness artefacts withdrawn from releases and endpoints?
Software supply chains	Inventories artefacts and checks integrity and provenance.	How can these controls cover learned rules and their review evidence?
Brevet reference loop	Records a capability lifecycle with review, evaluation, release, and recall.	Which additional controls are needed for complete enforcement?

Table 4: Conceptual positioning. These are differences of emphasis, not an exhaustive coverage audit or a claim that adjacent systems lack all governance mechanisms. Prototype coverage is detailed separately in Table 1.

Keys, identities, and review quality. The prototype uses per-workspace Ed25519 keys without rotation, and its promotion, release, and recall labels are not authenticated. The surrounding suite provides designs on which these controls can build. CHAP specifies profiles for participant identity, key rotation and revocation, signed evidence, and external audit-transparency receipts [11], while Metis structures mission-group review of captured knowledge [12]. The present integration mirrors envelopes into a CHAP workspace but does not yet bind Brevet’s authority operations to those participant identities, receipts, or review records. Integrating these controls and validating their enforcement remain deployment requirements under Table 1. Even authenticated approval establishes who authorised a change, not whether the reviewer deliberated well. Collusion and rubber-stamping remain organisational risks.

Recall, consent, and privacy. Identifier withdrawal is not digest-level exclusion, and recording rollback is not executing it. Derived capabilities, replicated inventories, cached prompts, and active sessions require an explicit propagation policy. Raw override text can also contain sensitive information. Consent metadata must be enforced during capture and reuse, with access control, minimisation, and a retention policy for underlying content. Preserving audit lineage does not require keeping every sensitive source indefinitely.

Scale and empirical scope. The evidence presented here comprises a synthetic lifecycle demonstration, a conversational integration exercise, conditional design arguments, and analytical results for the release gate. These establish the operation of the recorded lifecycle and the conditions under which its governance properties hold. Production adaptation quality, reviewer burden, and performance across larger capability inventories require longitudinal evaluation. The proposed benchmark and consented field studies provide complementary methods for investigating these questions.

10 Related Work

Table 4 positions Brevet alongside research on agent adaptation, memory, unlearning, and software supply chains. Brevet connects these concerns through a capability lifecycle that links operational correction evidence to human approval, evaluated releases, and explicit withdrawal records.

Self-improving agents. Skill acquisition [3], verbal self-correction [23, 24], prompt and pipeline optimisation [4, 5], and whole-harness optimisation [7, 8] demonstrate improvements under their respective evaluation conditions. Code-level evolution [6] and environment generation [9, 10] extend what may change in the loop. Brevet’s focus is complementary. It specifies when an improvement may be promoted into operational authority and how its release and withdrawal should be evidenced. Evaluation and version histories are valuable, but do not alone settle those authority decisions.

Memory for agents. Persistent-memory systems [1, 2] and long-horizon memory benchmarks [18, 35]

treat what to store and recall as a capability problem. The governed-memory model of Shahid et al. [12] additionally addresses provenance, applicability, and authority for tacit fragments captured by Metis. We introduce BREVET, a framework that extends this connection from governed memory to the lifecycle of learned agent capabilities.

Machine unlearning. Parameter-level unlearning [30, 31] seeks to remove specified training influence from model parameters; its cost and guarantees depend on the method. Harness-level recall is the complementary mechanism for explicit agent artefacts. Their inventory membership is addressable and inspectable; cessation of operational use still depends on enforcement and endpoint evidence.

Software supply chain. Lockfiles and bills of materials [19], signed provenance and supply-chain integrity [20–22], and advisory-driven recall directly inspired R3 and R5. BREVET applies these controls to explicit learned artefacts and their review evidence.

AI governance and documentation. Model cards and datasheets [37, 38] document static artefacts; regulatory frameworks [13, 15, 16] provide governance guidance or legal obligations according to their scope. Brevet specifies a runtime counterpart, machine-checkable change control over the learning itself, with CHAP [11] providing the evidence substrate and Shahid et al. [39] the broader composable-intelligence context.

11 Conclusion

Brevet places agent learning under change control through a capability lifecycle spanning proposal, human review, evaluation, release, and recall. It connects CHAP’s collaboration evidence and the tacit fragments captured by Metis to decisions about which learned capabilities may influence operational behaviour. Its central distinction is between evidence that supports a proposed change and authority to use that change.

The numerical analysis shows why aggregate improvement must be examined alongside task-level regressions, with independent observations and adequate sample sizes supporting estimates of regression risk. The design arguments establish the conditions under which approval precedes use and recalled capabilities remain excluded.

Future work will evaluate the loop through the proposed benchmark and consented field studies, measuring task outcomes, governance performance, and reviewer effort. The governing principle is simple. Agents propose changes, evaluations test their effects, and accountable humans authorise their operational use.

Availability

The reference implementation, schemas, specification, demonstration, conversational-assistant integration, and documentation are available at <https://github.com/BrightbeamAI/brevet> under Apache-2.0. The analysis in this paper was performed against version 0.1.0; the supplementary bundle records the source checksums it used (Appendix D).

References

- [1] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023. doi: 10.48550/arXiv.2310.08560. URL <https://arxiv.org/abs/2310.08560>.
- [2] Joon Sung Park, Joseph C. O’Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST)*, 2023. URL <https://arxiv.org/abs/2304.03442>.

- [3] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023. doi: 10.48550/arXiv.2305.16291. URL <https://arxiv.org/abs/2305.16291>.
- [4] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023. doi: 10.48550/arXiv.2310.03714. URL <https://arxiv.org/abs/2310.03714>.
- [5] Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*, 2023. doi: 10.48550/arXiv.2309.16797. URL <https://arxiv.org/abs/2309.16797>.
- [6] Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*, 2025. doi: 10.48550/arXiv.2505.22954. URL <https://arxiv.org/abs/2505.22954v3>. Version 3, revised 12 March 2026.
- [7] Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-harness: Harnesses that improve themselves. *arXiv preprint arXiv:2606.09498*, 2026. doi: 10.48550/arXiv.2606.09498. URL <https://arxiv.org/abs/2606.09498v3>. Version 3, revised 20 August 2026.
- [8] Hyunin Lee, Jinglue Xu, Jeffrey Seely, Donghyun Lee, Matei Zaharia, and Yujin Tang. Recursive harness self-improvement. *arXiv preprint arXiv:2607.15524*, 2026. doi: 10.48550/arXiv.2607.15524. URL <https://arxiv.org/abs/2607.15524v1>. Version 1, submitted 17 July 2026.
- [9] Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O. Stanley. Paired open-ended trailblazer (POET): Endlessly generating increasingly complex and diverse learning environments and their solutions. *arXiv preprint arXiv:1901.01753*, 2019. doi: 10.48550/arXiv.1901.01753. URL <https://arxiv.org/abs/1901.01753v3>.
- [10] Yucheng Shi, Zhenwen Liang, Kishan Panaganti, Dian Yu, Wenhao Yu, and Haitao Mi. Learning to build the environment: Self-evolving reasoning RL via verifiable environment synthesis. *arXiv preprint arXiv:2605.14392*, 2026. doi: 10.48550/arXiv.2605.14392. URL <https://arxiv.org/abs/2605.14392v1>. Technical report, work in progress.
- [11] Arsalan Shahid, Gordon Suttie, and Philip Black. Collaborative human-agent protocol (CHAP). *arXiv preprint arXiv:2606.09751v1*, Brightbeam AI, 2026. URL <https://arxiv.org/abs/2606.09751v1>.
- [12] Arsalan Shahid, Gordon Suttie, Philip Black, and Antonio Garzón-Vico. Tacit fragments: Operationalising tacit knowledge as a governed memory layer for agentic AI. *Preprints*, 2026. doi: 10.20944/preprints202608.0927.v1. URL <https://www.preprints.org/manuscript/202608.0927/v1>. Version 1, posted 13 August 2026.
- [13] International Society for Pharmaceutical Engineering. GAMP 5: A risk-based approach to compliant GxP computerized systems, second edition. ISPE, 2022.

- [14] U.S. Food and Drug Administration. 21 CFR part 11: Electronic records; electronic signatures. Electronic Code of Federal Regulations, Title 21, Part 11, 2026. URL <https://www.ecfr.gov/current/title-21/chapter-I/subchapter-A/part-11>. Current text consulted 20 September 2026; originally issued in 1997.
- [15] European Parliament and Council. Regulation (EU) 2024/1689 laying down harmonised rules on artificial intelligence (AI Act). Official Journal of the European Union, 2024. URL <https://eur-lex.europa.eu/eli/reg/2024/1689/oj/eng>. In particular Articles 12, 14, 17, 43(4), and 72.
- [16] Elham Tabassi. Artificial intelligence risk management framework (AI RMF 1.0). NIST AI 100-1, 2023. URL <https://nvlpubs.nist.gov/nistpubs/ai/nist.ai.100-1.pdf>.
- [17] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024. URL <https://arxiv.org/abs/2310.06770>.
- [18] Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. Evaluating very long-term conversational memory of LLM agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13851–13870. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.747. URL <https://aclanthology.org/2024.acl-long.747/>.
- [19] OWASP Foundation. CycloneDX bill of materials specification, version 1.5. Specification, 2023. URL <https://cyclonedx.org/docs/1.5/json/>.
- [20] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. in-toto: Providing farm-to-table guarantees for bits and bytes. In *28th USENIX Security Symposium*, pages 1393–1410. USENIX Association, 2019. URL <https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias>.
- [21] Open Source Security Foundation. SLSA: Supply-chain levels for software artifacts, version 1.0. Specification, version 1.0, 2023. URL <https://slsa.dev/spec/v1.0/>.
- [22] Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. Survivable key compromise in software update systems. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pages 61–72, 2010. doi: 10.1145/1866307.1866315.
- [23] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.
- [24] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*, 2023. doi: 10.48550/arXiv.2303.17651. URL <https://arxiv.org/abs/2303.17651>.
- [25] Michael McCloskey and Neal J. Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. In Gordon H. Bower, editor, *Psychology of Learning and Motivation*,

- volume 24, pages 109–165. Academic Press, 1989. doi: 10.1016/S0079-7421(08)60536-8. URL <https://www.sciencedirect.com/science/article/pii/S0079742108605368>.
- [26] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975. doi: 10.1109/PROC.1975.9939. URL <https://web.mit.edu/saltzer/www/publications/protection/>.
- [27] Erik Hollnagel. *Safety-I and Safety-II: The Past and Future of Safety Management*. Ashgate, 2014. URL <https://www.routledge.com/p/book/9781472423085>.
- [28] Simon Josefsson and Ilari Liusvaara. Edwards-curve digital signature algorithm (EdDSA). RFC 8032, RFC Editor, 2017. URL <https://www.rfc-editor.org/rfc/rfc8032>.
- [29] Anders Rundgren, Bret Jordan, and Samuel Erdtman. JSON canonicalization scheme (JCS). RFC 8785, RFC Editor, 2020. URL <https://www.rfc-editor.org/rfc/rfc8785>.
- [30] Yinzhi Cao and Junfeng Yang. Towards making systems forget with machine unlearning. In *IEEE Symposium on Security and Privacy (S&P)*, 2015. doi: 10.1109/SP.2015.35. URL <https://www.cs.columbia.edu/~junfeng/papers/unlearning/>.
- [31] Lucas Bourtole, Varun Chandrasekaran, Christopher A. Choquette-Choo, Hengrui Jia, Adelin Travers, Baiwu Zhang, David Lie, and Nicolas Papernot. Machine unlearning. In *IEEE Symposium on Security and Privacy (S&P)*, 2021. URL <https://arxiv.org/abs/1912.03817>.
- [32] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023. doi: 10.48550/arXiv.2308.08155. URL <https://arxiv.org/abs/2308.08155>.
- [33] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023. URL <https://arxiv.org/abs/2210.03629>.
- [34] C. J. Clopper and E. S. Pearson. The use of confidence or fiducial limits illustrated in the case of the binomial. *Biometrika*, 26(4):404–413, 1934. doi: 10.1093/biomet/26.4.404. URL <https://doi.org/10.1093/biomet/26.4.404>.
- [35] Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. LongMemEval: Benchmarking chat assistants on long-term interactive memory. In *International Conference on Learning Representations (ICLR)*, 2025. URL <https://openreview.net/forum?id=pZiyCaVuti>.
- [36] Marilyn Strathern. ‘Improving ratings’: Audit in the British university system. *European Review*, 5(3):305–321, 1997. URL <https://www.cambridge.org/core/journals/european-review/article/FC2EE640C0C44E3DB87C29FB666E9AAB>.
- [37] Margaret Mitchell, Simone Wu, Andrew Zaldivar, Parker Barnes, Lucy Vasserman, Ben Hutchinson, Elena Spitzer, Inioluwa Deborah Raji, and Timnit Gebru. Model cards for model reporting. In *Proceedings of the Conference on Fairness, Accountability, and Transparency (FAT*)*, 2019. URL <https://arxiv.org/abs/1810.03993>.

- [38] Timnit Gebru, Jamie Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna Wallach, Hal Daumé III, and Kate Crawford. Datasheets for datasets. *Communications of the ACM*, 64(12):86–92, 2021. URL <https://arxiv.org/abs/1803.09010>.
- [39] Arsalan Shahid, Gordon Suttie, and Philip Black. Composable digital intelligence. *Preprints*, 2026. doi: 10.20944/preprints202606.0854.v1. URL <https://www.preprints.org/manuscript/202606.0854/v1>.

A Evidence Envelope Kinds

Kind	Emitted by	Meaning
brevet.task	stage 1	one wrapped agent invocation
brevet.artefact	stage 1	an agent output attached to a task
brevet.override	stage 2	a human judgement: diff, rationale, tags
brevet.candidate	stage 3	a capability proposed by the delta engine
brevet.model_assist	stage 3	a logged model-drafting event (drafts only)
brevet.promotion	stage 4	a dawn-gate decision
brevet.eval_run	stage 5	a regression-suite execution
brevet.release	stage 6	a signed harness version transition
brevet.recall	stage 7	a capability recall notice

Table 5: Envelope kinds appended to the hash-linked evidence chain (Eq. 2), by loop stage. All envelopes are append-only and independently replayable.

B Demonstration Manifest

The five-layer manifest of the demonstration agent, signed by the release step with the signature block elided. The declarations are configuration metadata, not evidence that every policy is enforced.

```
agent: deviation_triage_assistant
version: 0.2.0
description: Demo: drafts deviation severity classifications
             (suggest-class only).
identity_policy:
  agent_id: deviation_triage_assistant
  owner: human:quality.owner@example.com
  mission_group: mission_group:right_first_time
  may: [draft_triage]
  may_not: [close_deviation]
prompt_architecture:
  system_prompt: Classify deviation severity; draft only.
cognitive_core:
  model_policy: {local_default: 'ollama:gemma4:12b', allowed_remote: []}
bindings:
  tools:
    read: [deviation_log.search]
    suggest: [draft_triage]
    act: []
    controlled_act: []
  capabilities_lock: capabilities.lock
runtime_safety:
  loop: {policy: react, max_iterations: 4}
  evidence: {ledger: 'file:./ledger.jsonl', capture_overrides: true}
  evals: {gate: conservative}
  dream: {enabled: true}
  release: {channel: trial}
```

C Command-Line Interface

Command	Effect
<code>brevet init</code>	scaffold <code>agent.yaml</code> and the <code>workdir</code>
<code>brevet demo</code>	run the whole loop on synthetic data, offline
<code>brevet dream</code>	mine the ledger into candidate capabilities
<code>brevet dawn</code>	list pending candidates, or apply one decision
<code>brevet release</code>	gate-check, sign, and release the next version
<code>brevet recall</code>	revoke a capability; flag affected releases
<code>brevet verify</code>	independently replay the evidence chain
<code>brevet status</code>	version, channel, capability counts, chain health
<code>brevet mcp</code>	serve the lifecycle over the Model Context Protocol

Table 6: The reference CLI. Every command is also available on the wrapped-agent object or corresponding MCP tools through shared lifecycle functions. Authentication and deployment enforcement remain separate concerns.

D Reproducing the Analysis

The supplementary source bundle contains `supplementary/reproduce_analysis.py` and the corresponding `analysis_results.json`. After installing the supplied Brevet source and its core dependencies, run the following command with a fresh output directory.

```
python supplementary/reproduce_analysis.py /path/to/brevet-main run-output
```

The script runs the synthetic demonstration, probes the implementation boundaries listed in Section 6, enumerates the sixteen four-case partition assignments, and computes the confidence bounds in Table 3. It records the Python and Brevet versions and SHA-256 checksums of the source files. No model, network access, or production data is required. Generated identifiers and timestamps vary; the reported counts, analytical results, and probe outcomes are the reproducibility targets. The supplied implementation is not modified.