Voder: Exploring Safe Artificial Superintelligent Agents

Libo Wang, Houde Liu*
VeriLoop Team
November 2026
wanglibo@sz.tsinghua.edu.cn
*Corresponding author: Houde Liu

ABSTRACT

Artificial superintelligence (ASI) raises a control question before its attainment: how can increasingly capable agents propose consequential changes without becoming the authority that accepts those changes? We present Voder as an exploration of safe, verifiable ASI agents, implemented and evaluated in software environments. VeriLoop-Governed Recurrence (VGR) separates candidate generation from evidence admission, native commitment, and workspace delivery. A trusted host binds approved paths, source preimages, and exact candidate identities; isolated checks establish candidate-specific evidence; Rust governance admits trusted receipts; and independent native confirmation precedes verified writeback. Journaled intent, replay fencing, and bounded contextual retrieval support continuity and recovery. The retained engineering record reports 318/318 component checks, 420/420 concurrency executions, 11/11 SDK recovery assertions, and three daemon-restart cases within their stated scopes. Synthetic-history selection reduced estimated context tokens by 93.3–98.7%, without measuring provider-billed savings. Fixed-denominator local reward audits under the author-reported DeepSeek V4.1 Flash 1017 configuration yielded 78.8%, 41.9%, and 45.5% on DeepSWE v1.1 and Terminal-Bench 3.0 and 4.0, respectively; a separate reference-assisted SWE-Marathon control yielded 45.0%. Interactive applications document delivered artifacts. These results establish a bounded engineering basis for studying externally governed acceptance, not ASI capability or comprehensive safety. We identify specification adequacy, complete mediation, trusted-component integrity, and invariant preservation across self-modification as conditions requiring further research.

Keywords: artificial superintelligence (ASI); ASI agents; verifiable acceptance; AI control; VeriLoop-Governed Recurrence (VGR); governed self-modification.

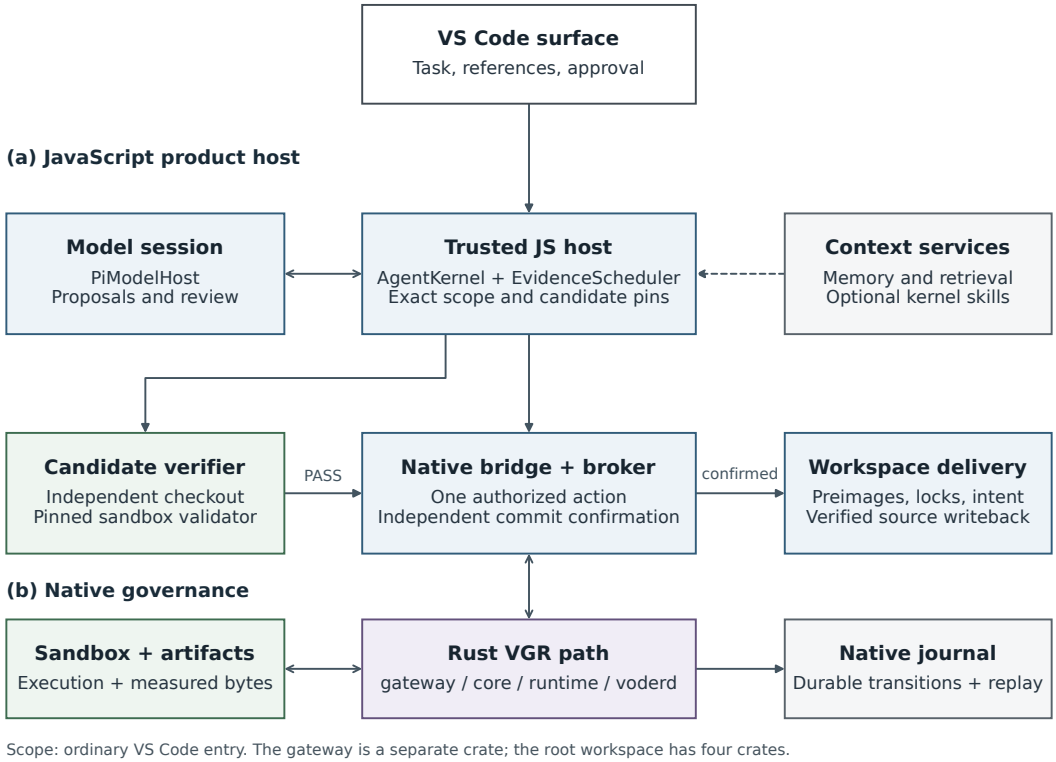


Figure 1: Current engineering foundation for exploring safe ASI agents. The implemented Voder software path separates proposal generation, candidate checking, native VGR qualification, and verified workspace delivery. Memory and retrieval supply context, not authority. This is the present experimental control substrate, not an attained ASI architecture; boxes denote responsibilities rather than one process per component.

Figure 1 presents Voder's current engineering foundation for investigating safe ASI-agent control. The implemented system operates on software tasks: its model proposes content, while the host binds scope and candidate checks, Rust VGR governs native acceptance, and independently confirmed authority precedes destination writeback. Context services support reasoning without issuing permits. This separation makes it possible to study whether more capable proposers can remain subject to externally maintained acceptance conditions. The diagram records the mechanisms available for that investigation; the ASI-agent objective is prospective, and the depicted components do not establish general superintelligence.

Contents

Abstract **1**

1 Introduction **4**

2 Related Work **5**
- 2.1 Frontier capabilities and the boundary of ASI claims . . . 5
- 2.2 Scientific discovery and recursive improvement . . . 5
- 2.3 Verifiable behavior and protected acceptance . . . 6
- 2.4 External control and evidence under adversarial behavior . . . 6
- 2.5 Practical foundations: Pi and contemporary agent harnesses . . . 7
- 2.6 DeepSeek Harness and the ordering of persistent effects . . . 7

3 Architecture and Algorithms **8**
- 3.1 From E2 supervision to Harness authority . . . 8
- 3.2 System decomposition and trust boundary . . . 9
- 3.3 Exact candidate identity and evidence-guided scheduling . . . 9
- 3.4 Native transaction and resource conservation . . . 12
- 3.5 Durable dispatch and uncertain-outcome fencing . . . 13
- 3.6 Trusted evidence admission and obligation closure . . . 15
- 3.7 Fresh meta-audit and independent commit confirmation . . . 17
- 3.8 Verified delivery and conservative restoration . . . 18
- 3.9 Final-source visual acceptance . . . 19
- 3.10 Bounded memory with preserved provenance . . . 21
- 3.11 Read-only capabilities without authority escalation . . . 22
- 3.12 Non-authoritative library extensions . . . 24
- 3.13 Composition and guarantee boundary . . . 26

4 Engineering Implementation **26**
- 4.1 Implementation lineage and integration sequence . . . 27
- 4.2 Build roots and deployment contract . . . 27
- 4.3 Pi adaptation and the model-session boundary . . . 29
- 4.4 Provider configuration and request continuity . . . 30
- 4.5 Task admission, scope preparation, and exact edits . . . 31
- 4.6 Native dispatch, sandbox execution, and observation . . . 31
- 4.7 Durable sessions and outcome reconciliation . . . 33
- 4.8 Verified delivery and original-project publication . . . 34
- 4.9 Context policy, skills, and engineering memory . . . 36
- 4.10 Read-only tools, retrieval, and planning aids . . . 37
- 4.11 Reference inputs and final-source visual inspection . . . 37
- 4.12 Operator interfaces, diagnostics, and release scope . . . 38

5 Empirical Evaluation **39**
- 5.1 Evaluation protocol and evidence provenance . . . 39
- 5.2 Component verification and concurrency stress . . . 40
- 5.3 Native authority and delivery validation . . . 41
- 5.4 Crash recovery and durable-state replay . . . 42
- 5.5 Context selection and estimated token reduction . . . 42
- 5.6 Context-policy ablation . . . 43
- 5.7 Isolated API latency . . . 44
- 5.8 Audited benchmark outcomes . . . 44
- 5.9 Validity and attribution boundaries . . . 45

6 Interactive 3D Application Case Studies **46**
- 6.1 Voder capability demonstration: cycling pelican . . . 46
- 6.2 Scripted reusable-booster recovery . . . 47
- 6.3 Shader-based black-hole visualization . . . 47
- 6.4 Geometric wormhole exploration . . . 48
- 6.5 Interactive racing and state coherence . . . 48
- 6.6 Cross-case assessment and scope . . . 49

7 Limitations and Future Work **49**
- 7.1 Extending the substance of acceptance obligations . . . 49
- 7.2 Deployment integrity and effect-specific mediation . . . 50
- 7.3 Interruption coverage and cumulative control evidence . . . 50
- 7.4 Controlled self-modification and invariant preservation . . . 50
- 7.5 A constructive research vision for safe ASI agents . . . 51

8 Conclusion **51**

References **52**

A Implementation Traceability and Authority Boundaries **54**

B Evaluation Evidence and Reproduction Boundaries **54**

C Interactive Demos and Reproducibility **55**

1 Introduction

On February 5, 2026, OpenAI reported that early GPT-5.3-Codex versions assisted their own training, deployment, and evaluation [38]; Anthropic released Claude Opus 4.6, advancing sustained autonomous coding [39]. Anthropic's April 7 Claude Mythos Preview announcement and related AI-training optimization research further illustrated AI-assisted AI development [40, 41]. These advances motivate recursive self-improvement (RSI): AI helps improve successor systems that may drive further advances, although sustained autonomous recursion remains unproven. Artificial superintelligence (ASI) would consistently surpass human experts across broad, important cognitive domains [1]; an *ASI agent* would apply this competence through planning, tools, and consequential action. Voder investigates externally governed control in software environments without claiming ASI has been attained.

This capability threshold distinguishes ASI from artificial general intelligence (AGI). AGI concerns broadly transferable competence across diverse tasks, whereas ASI further entails pervasive superiority over human experts. Generality, performance, and autonomy are separate dimensions: excelling in one domain does not establish ASI, and even a highly capable system need not receive unrestricted operational authority [1]. Recursive self-improvement is one plausible route to ASI, not a defining property or a demonstrated prerequisite [37].

Broader machine competence could accelerate scientific discovery, engineering design, and the evaluation of new hypotheses; AlphaEvolve already demonstrates a narrower form of this opportunity [2,3], while an independent 2026 assessment finds evidence of AI-assisted research acceleration under substantial uncertainty [4]. The same growing ability to plan and act can amplify errors in objectives, supervision, or environmental assumptions. A flawed experiment can shape subsequent research, a mistaken software update can propagate through dependencies, and an external action can cause effects that cannot be undone by reverting code. Objective misspecification, reward exploitation, distribution shift, and incentives against intervention are distinct risks; none requires a conscious or hostile agent [5–8]. Useful capability and reliable control must therefore be established independently.

The *core gap* is a divergence between capability growth and enforceable control: improving an agent's reasoning, planning, or ability to modify systems does not necessarily improve the alignment of its objectives or the safeguards that constrain its actions. Recursive improvement exposes a precise technical instance, the *safety-invariant preservation gap*. Let s_t be the retained system state, including its operative code and control configuration, and let $\mathcal{I}$ denote a protected safety invariant. A safe accepted transition would have to satisfy $\mathcal{I}(s_t) \wedge \text{Accept}(s_t, s_{t+1}) \Rightarrow \mathcal{I}(s_{t+1})$ under explicitly stated assumptions. Yet a candidate may change the very code, validator, permissions, or acceptance policy used to judge its successor. Passing a test for one version cannot by itself establish that the protected constraint survives the next version, and an agent-controlled evaluator cannot provide independent assurance. The resulting engineering obligations are concrete: bind each candidate to the exact prior state and a separately owned contract; obtain trustworthy, candidate-specific evidence; prevent unqualified effects and ambiguous retries; and preserve the integrity and applicability of the enforcement boundary across accepted changes. Complete mediation and separation of privilege motivate this architecture [9], while AI-control and formal-safety research make clear that its guarantee remains conditional on sound specifications, valid evidence, and trusted enforcement [6, 10]. This is a central research problem for governed recursive improvement, not an established single cause of every possible ASI failure.

Voder investigates the enforceable part of this gap through VeriLoop-Governed Recurrence (VGR), adapted from the VeriLoop team's E2 framework to an externally governed software execution path. A model may propose edits, but the host fixes approved scope, source preimages, and candidate identity; isolated checks produce candidate-bound evidence; Rust governance admits trusted observations and conducts fresh audits; independent native confirmation precedes separately verified workspace delivery. Failed or uncertain transitions cannot self-authorize acceptance or replay. This presently

enforces a bounded proposal-to-persistence boundary, not cross-generation safety of self-modifying ASI systems. The deployed gateway checks a pinned validator-success obligation, which does not prove complete semantic correctness or that its specification captures all human interests. VGR's contribution is thus an implemented, evidence-governed acceptance substrate and a concrete research basis for examining whether stronger specifications and protected validators can preserve safety invariants under future controlled self-modification.

2 Related Work

The control question introduced above connects three bodies of work: evidence about frontier capabilities, methods for verifying or constraining consequential behavior, and practical agent execution systems. The review cutoff is November 7, 2026. We distinguish developer-reported results, independent assessments, theoretical guarantees, and source-level observations; none is treated as evidence that a broadly superintelligent agent has already been demonstrated.

2.1 Frontier capabilities and the boundary of ASI claims

Capability assessment first requires separating superhuman task performance from general superintelligence. The Levels of AGI framework distinguishes performance, generality, and autonomy, providing a useful vocabulary without establishing a universally accepted ASI certification test [1]. Consequently, a frontier system's research or benchmark performance can motivate control research without qualifying the system as ASI.

Recent public evidence shows substantial but uneven progress. Google DeepMind's September 2026 Gemini 3.8 Flash model card reports 73.7% on DeepSWE v1.1, 19.1% on Terminal-Bench 4.0, and 54.9% on HLE-Verified under its published evaluation configurations [11]. These results show advanced software and multidisciplinary reasoning capabilities alongside considerable residual failure. They are provider-reported measurements, not matched comparisons with Voder's retained-reward audits; differing configurations and denominators prevent direct ranking. METR's task-horizon methodology supplies a complementary view of autonomous task completion, defined relative to human expert task duration and a specified success probability [12]. Such measurements characterize selected task distributions, rather than reliable operation across arbitrary environments.

These distinctions set the evidential boundary for the present report. The reviewed public record supports increasingly capable general-purpose systems and specialized research contributions. It does not establish the broad, sustained superiority or comprehensive control guarantees required by the prospective ASI-agent definition used here. Voder's software experiments therefore investigate a control mechanism relevant to that objective; they do not measure distance to ASI.

2.2 Scientific discovery and recursive improvement

Research acceleration makes this control mechanism relevant before general superintelligence is established. AlphaEvolve combines language-model proposals, automated evaluation, and evolutionary search to discover and optimize algorithms [2]. Its May 2026 update reports applications spanning computing infrastructure, mathematics, genomics, and physical-science computation [3]. These are concrete examples of machine-generated contributions to research and engineering, with usefulness assessed through domain-specific evaluation rather than a universal intelligence criterion.

The November 5, 2026 METR assessment of Anthropic provides a more recent, independent perspective on AI-assisted AI development [4]. METR judges that progress on well-specified tasks was likely occurring at approximately twice the earlier historical rate, with high uncertainty, and attributes a majority of the acceleration to AI use. It also reports that contributions remained primarily assistive

and that substantial supporting evidence was redacted. This supports investigating feedback between AI capability and AI development, while leaving the strength, sustainability, and autonomy of that feedback unresolved.

Self-modification is a more specific mechanism than research assistance. The Gödel Machine proposes self-rewrites conditional on proofs of usefulness relative to encoded axioms and utility [13]. The Darwin Gödel Machine instead modifies coding-agent implementations, evaluates variants empirically, and maintains an archive for further exploration [14]. More recent harness-policy optimization studies likewise examine agent evolution [15]. These approaches motivate Voder's future self-modification experiments, but performance improvement under an evaluation procedure does not by itself preserve safety constraints. VGR addresses the authority and identity conditions under which a proposed change may be retained; sustained capability improvement through that path remains to be demonstrated.

2.3 Verifiable behavior and protected acceptance

The theoretical basis for protected acceptance predates language-model agents. Saltzer and Schroeder's protection principles include complete mediation, fail-safe defaults, and separation of privilege [9]. Their relevance is architectural: a decision can constrain an effect only if the protected operation actually passes through the enforcing boundary. Voder applies this reasoning to the acceptance of generated software changes under a trusted host.

Proof-carrying code makes the verification relationship explicit: an untrusted producer supplies code and a machine-checkable argument that it satisfies the receiving host's stated safety policy [16]. The approach demonstrates how powerful or untrusted generation can be separated from comparatively small trusted checking. Its guarantee is policy-relative; it does not establish that an arbitrary policy captures every human interest. Voder's current runtime evidence is an authenticated account of configured checking, rather than a general proof-carrying-code implementation.

Guaranteed Safe AI develops a related assurance framework around a world model, a safety specification, and a verifier [10]. This provides a useful conceptual placement for VGR: governance can enforce an acceptance decision, but the strength of that decision depends on the specification and the evidence interpreter. Our use of *verifiable* therefore concerns explicit, checkable acceptance obligations and their binding to retained artifacts. It does not mean that all model reasoning, all generated behavior, or every downstream consequence has been formally verified.

2.4 External control and evidence under adversarial behavior

Explicit obligations still require evaluating whether oversight remains effective when a model acts against them. AI Control studies protocols that combine stronger untrusted models, weaker trusted models, and limited trusted human work, and evaluates them against intentional subversion in programming tasks [6]. It establishes an empirical research direction for control without assuming that the system reliably intends to comply. VGR shares the separation between proposal and acceptance, while implementing artifact identity, native decision authority, and persistence conditions rather than the paper's model-monitoring protocols.

The limits of this separation are equally important. Concrete Problems in AI Safety distinguishes misspecified objectives, expensive supervision, unsafe exploration, and distribution shift [5]; the Off-Switch Game analyzes how objective uncertainty affects incentives to permit intervention [7]. These are complementary problems, not failures that a commit gate automatically solves. OpenAI's October 29, 2026 description of frontier monitoring combines activation-based screening, model review, investigation, external telemetry, and human response [8]. Monitoring supplies observations and escalation; a pre-effect authorization boundary supplies a distinct enforcement function. VGR could

participate in such layered defenses, but the present tests do not establish resistance to an adversarial superintelligent proposer or a compromised trusted host.

2.5 Practical foundations: Pi and contemporary agent harnesses

An enforcing boundary must integrate with a functioning execution substrate. Pi supplies Voder's direct model/session lineage: the provided archive retains version 1.1.0 source, upstream attribution, and licensing [17]. Session execution, provider conversion, streaming, and context compaction are reused, while Voder supplies task scope, evidence scheduling, native governance, and controlled delivery. Pi's generic tool-result interface does not itself impose Voder's exact-candidate native-commit contract; that responsibility belongs to the integrating system.

Other harnesses provide complementary precedents. Codex's App Server separates threads, turns, typed items, and approval requests, while its platform work distinguishes client surfaces from agent execution [18–20]. Claude Code's long-running-agent guidance emphasizes progress artifacts, explicit evaluation, and continuity across bounded sessions [21, 22]. ZCode exposes a coding runtime through multiple clients [23]. These contributions inform lifecycle design and usable integration, but a completed conversational turn or restored context does not independently certify the bytes ultimately delivered.

Research systems further clarify the supporting layers. SWE-agent and OpenHands investigate execution interfaces and agent environments; Agentless separates repair stages and patch validation [24–26]. MemGPT and long-context studies motivate selective contextual access, while recent skill and concurrency studies show that orchestration benefits depend on workload and communication structure [27–30]. Voder builds on these practical lessons without equating better context or more parallel computation with stronger acceptance authority. Their role is to make capable execution feasible; VGR's role is to govern retention within its declared boundary.

2.6 DeepSeek Harness and the ordering of persistent effects

The need to distinguish execution from acceptance is concrete in the DSH comparison retained from the original engineering analysis. DeepSeek Harness documents a plugin-composed agent architecture [31]. The report's recorded static inspection concerns snapshot `fcc0ad462172` (full identity in the cited source), rather than every DSH version or possible plugin configuration [32].[1]

In those reviewed default paths, source-presence and version guards precede the write, but `applyWriteTool` performs the write before the shared pipeline evaluates `postExecute`. The recorded rejection branch returns an error without itself restoring the previous bytes. Rejection of the tool report therefore does not establish rejection of the persistent mutation. Likewise, conversational completion does not impose a final-artifact acceptance predicate. This is a substantive limitation of the inspected composition: a post-effect verdict cannot establish that rejected content never reached the destination.

The observation concerns an enforceable ordering property, not a claim that DSH cannot be extended or that the gap proves a general ASI vulnerability. Voder implements a different composition for its controlled delivery: check a bound candidate, admit trusted evidence, independently confirm native commitment, and verify destination writeback. The resulting distinction links practical harness engineering to the broader research question of preserving external acceptance authority as agent capability and self-modification increase.

[1] Recorded inspected paths: `packages/fs/tool-fs/src/write.ts`, `packages/fs/fs-observation-policy/src/index.ts`, `packages/core/tools/src/pipeline.ts`, and `packages/core/agent-loop/src/agent.ts`. This revision retains the version-pinned observation; it does not report a newly reproduced execution or exploit.

3 Architecture and Algorithms

The authority-and-effect gap identified in Section 1 becomes an engineering problem at the boundary between a candidate proposal and retained state. Voder addresses that gap through an evidence-governed sequence: isolate and identify the candidate, authorize its execution, admit trusted observations, independently confirm native commitment, and transfer only the qualified bytes. VGR is the governance logic across these stages, rather than a model instruction to behave safely. Its relevance to ASI-agent research is that greater proposal capability need not confer greater acceptance authority. This chapter derives that property for the current software boundary; extension to broader effects and self-modification remains an explicit research task. The contribution is the composition of established systems mechanisms into this enforced path [9].

3.1 From E2 supervision to Harness authority

VGR originates in the VeriLoop team's E2 framework, where externally verified acceptance decisions govern retained state and supply labels for post-training. For a fixed obligation schema, E2 compares an incumbent rank vector $\boldsymbol{r}_t \in \mathbb{N}_0^m$ with a candidate $\boldsymbol{r}'_t$ using

$$\boldsymbol{r}'_t \prec_{\mathrm{P}} \boldsymbol{r}_t \iff (\forall j \in \{1, \dots, m\},\ r'_{t,j} \le r_{t,j}) \wedge (\exists j,\ r'_{t,j} < r_{t,j}).$$

Here $\mathbb{N}_0$ denotes the nonnegative integers, the subscript P labels the protected order, t is a proposal-attempt index, m is the positive number of fixed protected coordinates, j indexes those coordinates, and a smaller rank means fewer remaining violations. The indicator $\chi_t = \mathbf{1}[\boldsymbol{r}'_t \prec_{\mathrm{P}} \boldsymbol{r}_t]$ determines retention and the associated acceptance label. E2 binds the artifact, latent state, verification result, and digest as one retained bundle; rejected proposals cannot replace that bundle.

Those acceptance labels also connect the discrete controller to E2's differentiable supervision. For a same-incumbent accepted/noncommittable pair (y^+, y^-) under conditioning record c, its preference objective takes the form

$$\mathcal{L}_{\text{pref}}(\phi) = \mathbb{E}_{(c,y^+,y^-)} \operatorname{softplus}\big(\mu - \big[\ell_\phi(y^+ \mid c) - \ell_\phi(y^- \mid c)\big]\big)\,.$$

Here y^+ and y^- are candidate token sequences whose protected-order labels are checked against the same retained state and contract; c contains the permitted task/state/evidence conditioning. ℓ_ϕ is target-sequence log-probability with adaptation parameters ϕ, $\mu > 0$ is the preference margin, $\operatorname{softplus}(z) = \log(1 + \exp z)$ for scalar z, and $\mathbb{E}$ averages valid pairs.

The objective encourages the proposer to prefer verified improvements; it does not make every sampled proposal valid. The transferable mechanism is therefore the external acceptance rule that supplies the labels, rather than the numerical optimizer itself.

For control, the distinction is decisive: learning can improve the distribution of proposals, while an acceptance rule governs which proposals may alter protected state. Voder carries the second function into the execution harness. This connects E2 to the research question in Section 1 without assuming that an improved proposer becomes its own reliable safety judge.

Moving this principle into a coding harness changes the controlled object and the point of enforcement. A repository edit is an externally observable effect, whereas a latent displacement belongs to the proposal computation. Voder consequently controls candidate files, execution identities, admitted evidence, and delivery records, without optimizing this loss online or modifying model weights or hidden states. It preserves the separation between proposal and persistence authority, but does not import E2's protected-rank descent as a theorem about arbitrary software edits. In particular, the current gateway closes one pinned validator-success obligation rather than evaluating a general multi-coordinate semantic progress vector.

Three design choices make that relocation operational. First, proposal identity includes both the exact replacement and the source preimage, so a check cannot silently authorize a different candidate. Second, only a trusted execution path can construct evidence eligible for native admission; model text, remembered PASS results, and ordinary tool reports remain inputs to reasoning. Third, acceptance is split into native qualification and workspace delivery, so an uncertain execution or interrupted writeback is reconciled at its own boundary instead of being disguised by another execution. Sections 3.3–3.8 derive these conditions from the implemented transitions.

3.2 System decomposition and trust boundary

These authority boundaries are enforced by distinct native responsibilities. The gateway parses a bounded proposal and compiles the operator-pinned policy into a frozen contract. Core checks action, budget, obligation, and evidence rules; Runtime owns sandbox execution, measured consequences, artifacts, and durable dispatch eligibility. The daemon transition code appends and reconstructs the native journal, while Protocol carries typed requests, state, and replay results. Durable journal ordering and crash-consistency concerns are established systems problems; Voder's specific event contract and native qualification remain its own implementation [33, 34]. The independent observer uses that protocol to inspect durable state after the producer releases its writer lock. None of these roles is replaced by successful JSON decoding or by the gateway's own completion message.

This boundary assumes trusted host configuration, adapter code, native binaries, validator registration, and filesystem enforcement. Source content, retrieved passages, conversation history, skill material, and model outputs are untrusted data. A hash binds bytes under the collision-resistance assumption; it neither establishes their truth nor protects against a malicious host rewriting the entire record [35]. The following algorithms formalize checks within this trust boundary, not an externally attested execution environment.

Within that boundary, Figure 2 maps the general acceptance problem to four implemented control obligations. Candidate identity limits substitution, trusted admission separates assertions from observations, independent replay preserves the meaning of prior effects, and destination checks carry authorization to retained bytes. These are complementary obligations: evidence provenance without byte identity, or an authorized commit without checked delivery, would leave a different gap. The mapping states an engineering control argument, not a complete taxonomy of ASI risk.

Notation. $H(b)$ denotes lowercase hexadecimal SHA-256 of byte string b; $\mathrm{UTF8}(s)$ and $|s|_8$ denote UTF-8 encoding and encoded byte length of string s. The operator $\|$ concatenates exact bytes, dec produces base-10 ASCII, and $\perp$ denotes an absent value. JS is the source's exact `JSON.stringify` encoding; RustJSON is the native protocol value encoder, not an arbitrary interchangeable serializer. $\mathbf{1}[\cdot]$ is a Boolean indicator, clone creates private trial state, and Persist includes the required append and flush/fsync. Symbols are local to their algorithm or derivation; vectors are bold italic, while named operators, fields, states, and descriptive subscripts are upright.

3.3 Exact candidate identity and evidence-guided scheduling

The first enforcement point binds proposed work to an approved starting state. As Figure 3 shows, repository inspection may precede write approval, but obtaining write scope fixes the operator-approved paths and their preimage hashes. A candidate is data until the verifier and subsequent gates accept its exact identity. For algorithm A01, let $p = (r, o, x)$ contain relative path r, expected original digest o, and replacement text x. Single-file and batch identities use different array encodings:

$$d(p) = H(\mathrm{UTF8}(\mathrm{JS}([r, o, x]))) \,. \tag{3.1}$$

$$\begin{aligned} P &= (p_1, \ldots, p_n), \qquad 2 \le n \le 16, \\ d(P) &= H(\mathrm{UTF8}(\mathrm{JS}([[r_i, o_i, x_i]]_{i=1}^{n}))) \,. \end{aligned} \tag{3.2}$$

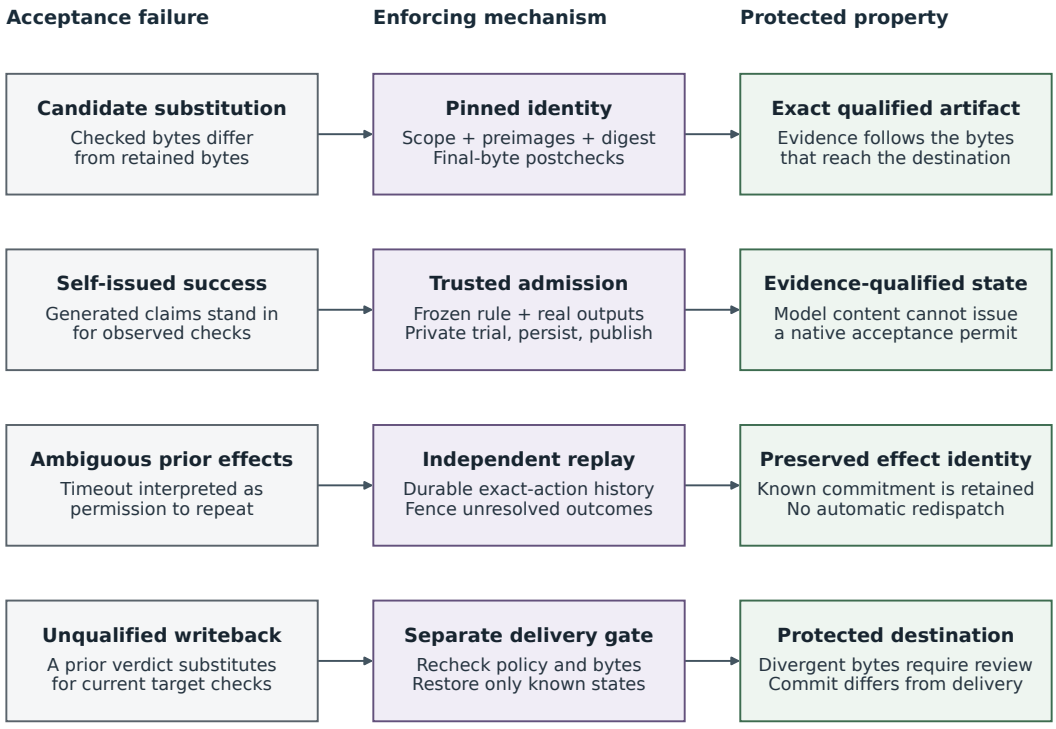


Figure 2: Acceptance failures and their implemented VGR controls. Each row links a concrete failure pathway to the enforcing mechanism and the property it protects under the trusted-host contract. The right column concerns current software-state transitions. Extending the same obligations to broader effects or recursive self-modification requires additional specifications and evaluation.

In the batch, P is the ordered member sequence, p_i its ith member, and n the number of targets. Members must be distinct, sorted existing paths; the implemented batch bound is $2 \le n \le 16$. For source checking, a single-file proposal is treated as a one-member sequence without changing its single-file digest encoding. With $\mathcal{R}_{\text{op}}$ the approved ordered path sequence and $W[r_i]$ the live source bytes, the gate is

$$\begin{aligned} G_{\text{source}}(P) = \ & \text{ValidCandidate}(P) \wedge [r_i]_{i=1}^{n} = \mathcal{R}_{\text{op}} \\ & \wedge \bigwedge_{i=1}^{n} [H(W[r_i]) = o_i]. \end{aligned} \tag{3.3}$$

ValidCandidate denotes the exact schema, authorized path, lowercase 64-hex preimage, nonempty UTF-8 replacement, and applicable batch-shape checks. Ordered equality prevents an approved scope from being expanded or reordered. The verifier pins its command, checks actual output hashes and sandbox identity, and runs against an independent checkout. A matching PASS is necessary for ordinary evidence-mode submission, but is not a native execution permit.

This identity binding constrains a specific route from persuasive output to uncontrolled effect: a check on one artifact cannot silently qualify another artifact or a changed starting state. The check remains attached to exact bytes and scope even when the model revises its explanation. What those bytes mean is determined by the configured obligations; the digest supplies identity, not semantic safety.

Preserving that distinction also requires controlling how the agent spends its remaining opportunity to verify and submit. Algorithm A02 shares one host call ledger across exploration, verification, and final action. Let M be the configured stage call limit, c the cumulative call count, c_0 the count at stage start, and V the optional verification cap. Remaining capacity R, fallback V_0, and reserve Q

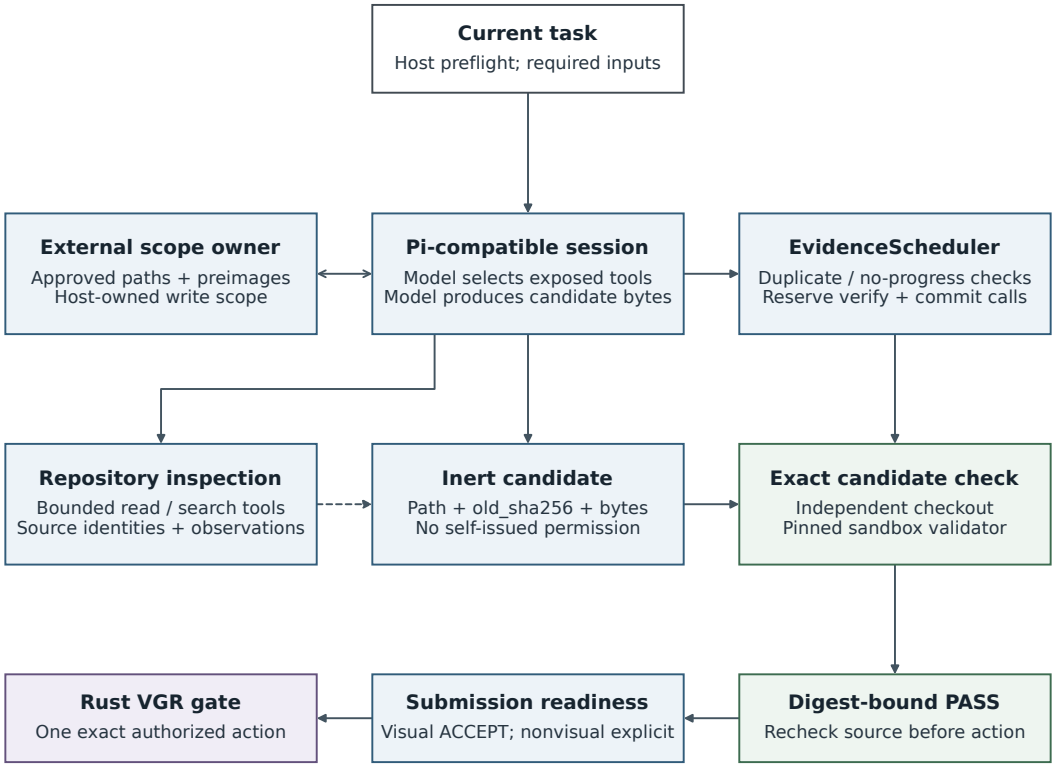


Figure 3: Host control and candidate admission. External scope ownership and source pins keep the proposal inert until exact-candidate checks qualify submission. A model-generated PASS cannot issue native authority. Verification and submission capacity remain reserved for the enforced path.

are

$$\begin{aligned} R &= \max(0, M - (c - c_0)), \\ V_0 &= \begin{cases} 12, & V = \bot, \\ V, & V \neq \bot, \end{cases} \\ Q &= \max(2, \min(V_0 + 1, \lfloor M/4 \rfloor)) \,. \end{aligned} \tag{3.4}$$

A proposed repository read moves the scheduler to finalization when either the no-progress count u reaches threshold U or $R \leq Q$. During finalization, a focused repair read requires a remaining diagnostic credit $z > 0$ and $R > 2$; verification requires $R > 1$. Each accepted call is charged before asynchronous work. Only distinct completed FAIL diagnostics can fund repair reads, with at most two such reads per task; repeated reads at the same evidence revision do not create progress.

The scheduler records a candidate as `OUTCOME_UNKNOWN` before asynchronous verification and consumes its native-attempt identity before dispatch. A failed stale-source precheck can release an unexecuted candidate probe; an uncertain remote check does not authorize automatic replay. Advancing a configured model stage resets stage expenditure and exploration pressure while retaining checked-candidate and consumed-action identities. Public retrieval has separate duplicate and no-progress accounting: it cannot replenish native verification capacity, mint repair credits, or reset an attempted action.

These reservations also control avoidable expenditure. A previously probed candidate digest is refused before its verifier runs again, and an already attempted action cannot be redispatched to obtain a more convenient outcome. Stored exact-candidate PASS supports the subsequent submission check within the task; it is not a cross-task verdict cache and does not replace Rust's independent validators.

Thus A02 spends computation on a changed proposal or an unobserved outcome, while retaining capacity for the checks required to close the task.

3.4 Native transaction and resource conservation

An exact host PASS permits the proposal to be submitted for native governance, not to bypass it. The gateway first requires a clean original Git repository, creates an owned candidate worktree, verifies every preimage, and rejects no-op replacements. It checks trusted sandbox availability before preparation so that an unavailable dependency does not strand a newly prepared action. One native action may carry a fixed multi-file batch; it is not restricted to a single replacement.

Figure 4 exposes the resulting state order. `PROPOSED` contains inert bytes; `VGR_PREPARED` seals intent and reservation without execution. Durable dispatch precedes observation of measured consequences, budget settlement precedes evidence admission, and meta-audit precedes a separately persisted final commit. Failures stop advancement at their own boundary. In particular, neither observation nor settlement grants delivery authority.

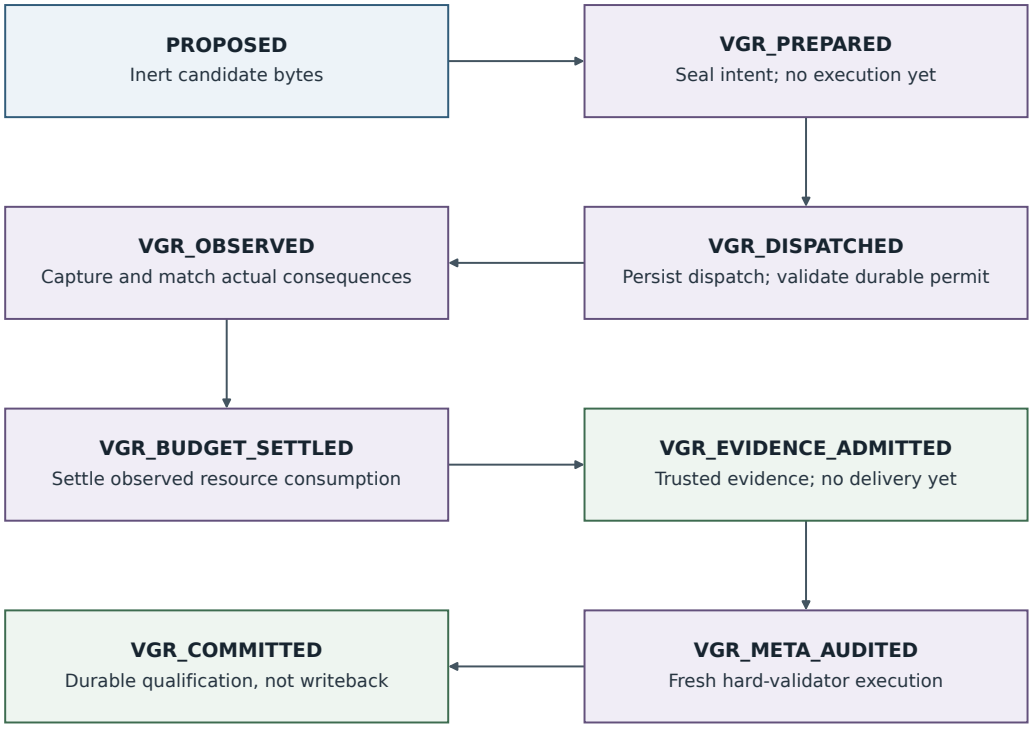


Figure 4: Native VGR transaction state machine. Preparation, dispatch, observation, evidence admission, and commitment carry different permissions. Each state advances only under its own checks; unknown effects stay fenced. Final native commitment is a qualification preceding separately verified workspace delivery.

Resource accounting constrains execution before its effects are incurred. In A03, let $\boldsymbol{t}$, $\boldsymbol{s}$, $\boldsymbol{r}$, and $\boldsymbol{a}$ be the total, spent, reserved, and available eight-coordinate resource vectors. Then

$$\boldsymbol{a} = \boldsymbol{t} - \boldsymbol{s} - \boldsymbol{r}, \qquad \boldsymbol{s} + \boldsymbol{r} \preceq \boldsymbol{t}. \tag{3.5}$$

The coordinates are steps, tokens, model calls, tool calls, wall time in milliseconds, risk microunits, active-context tokens, and API cost in millionths of USD. Risk is a policy score, not a probability. For reservation identifier k, phase ϕ_k, reserved upper bound $\boldsymbol{u}_k$, proposed bound $\boldsymbol{u}$, and measured

consumption $\boldsymbol{v}$, the updates are

$$\begin{aligned}
\text{Reserve}(k,\boldsymbol{u}):\quad & \mathbf{0} \prec_{\neq} \boldsymbol{u} \preceq \boldsymbol{a},\\
& \boldsymbol{r}' = \boldsymbol{r} + \boldsymbol{u}, \quad \phi_k' = \text{Held};\\
\text{Settle}(k,\boldsymbol{v}):\quad & \phi_k = \text{Dispatched}, \quad \boldsymbol{v} \preceq \boldsymbol{u}_k,\\
& \boldsymbol{s}' = \boldsymbol{s} + \boldsymbol{v}, \quad \boldsymbol{r}' = \boldsymbol{r} - \boldsymbol{u}_k;\\
\text{Cancel}(k):\quad & \phi_k = \text{Held}, \quad \boldsymbol{r}' = \boldsymbol{r} - \boldsymbol{u}_k.
\end{aligned} \tag{3.6}$$

$\preceq$ is componentwise ordering; $\mathbf{0} \prec_{\neq} \boldsymbol{u}$ means nonnegative and not identically zero. A prime denotes post-update state. Dispatch changes a held reservation to `Dispatched`; settlement and cancellation end in `Settled` and `Cancelled`, respectively. An identifier can be reused only for the same bound while still held. Dispatched work cannot be cancelled as though it never occurred, and an over-bound charge is refused for reconciliation rather than clamped.

Checked unsigned 64-bit arithmetic and these preconditions preserve $\boldsymbol{s} + \boldsymbol{r} \preceq \boldsymbol{t}$: reservation adds only available capacity; settlement replaces the reserved bound by a no-larger measured charge; cancellation removes only a held bound. This is a ledger conservation argument, not evidence that every adapter meters every coordinate. The current gateway reserves and settles steps, tool calls, and wall time for its fixed action; the other generic dimensions do not establish product-wide metering.

The control contribution of this ledger is prospective resource authorization: execution cannot justify its own earlier expenditure after the fact. This complements acceptance checks, but does not convert the policy's risk score into a measured probability of harm. Broader resource and effect budgets require appropriately instrumented adapters before they can strengthen the claim.

3.5 Durable dispatch and uncertain-outcome fencing

Budget feasibility alone cannot show whether an external effect has already been authorized. Voder therefore separates the host's dispatch intent from the native event history, as shown in Figure 5. The broker rejects concurrent and previously attempted idempotency keys, records unknown outcome before crossing the native boundary, and invokes the durable host checkpoint callback with the exact action identity. Native code independently records preparation and dispatch under its exclusive journal writer lock.

A04 binds that native authorization to a verifiable journal prefix. Let j be a one-based sequence number, b_j the native event body, and h_j its hash. The predecessor $h_0 = 0^{64}$ contains 64 ASCII zero characters:

$$\begin{aligned}
& h_0 = 0^{64},\\
& h_j = H\big(\texttt{voder-ledger-v1\textbackslash n} \,\|\, \text{dec}(j) \,\|\, \texttt{\textbackslash n}\\
&\qquad \|h_{j-1}\| \texttt{\textbackslash n} \,\|\, \text{UTF8}(\text{RustJSON}(b_j))\big).
\end{aligned} \tag{3.7}$$

The displayed `\n` represents a newline byte. ValidChain checks contiguous sequence, predecessor, and recomputed hash for every record. With N_{prepare} and N_{dispatch} the counts of qualifying exact-action records,

$$\begin{aligned}
G_{\text{dispatch}} = \ & \text{SafeJournalFile} \wedge \text{ValidChain}\\
& \wedge N_{\text{prepare}} = 1 \wedge N_{\text{dispatch}} = 1\\
& \wedge \text{LastIsExactDispatch}\,.
\end{aligned} \tag{3.8}$$

SafeJournalFile includes the production file, permissions/link, and writer-lock checks. Qualifying records match session, run, action, reservation, frozen contract, and workspace; preparation records execution as unauthorized, while dispatch records execution authorization with delivery still disallowed. LastIsExactDispatch requires the target dispatch to be the last durable event, even if a later event would concern another run.

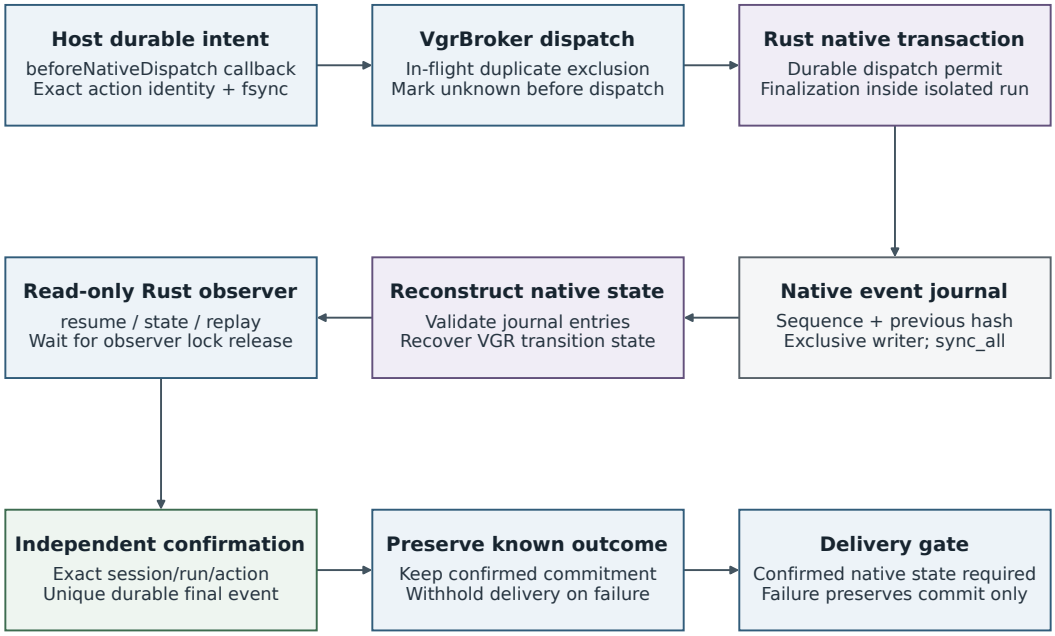


Figure 5: Durable dispatch and independent confirmation. The host preserves attempted identity, the native journal records governed transitions, and a separate observer reconstructs the exact committed outcome. Reporting failures cannot renew an action identity or substitute for checked destination delivery.

Algorithm 1 Unique current durable dispatch eligibility (A04)

Inputs: Plain private journal, prepared action identity, daemon writer lock.

```
previous ← 64 zero characters; sequence ← 0                                  Initialize
prepared ← false; dispatches ← 0; last_exact ← false
for each complete bounded journal record:
    require sequence increments by one
    require predecessor and recomputed event hash match                      Check integrity
    previous ← verified hash; sequence ← current sequence                    Advance cursor
    last_exact ← false
    if run/action do not match the prepared action: continue                 Match action
    if type is preparation or dispatch:
        require session, reservation, contract and workspace match           Bind identities
    if type is preparation:
        require not prepared and dispatches = 0
        require execution_authorized = false                                 Inert proposal
        prepared ← true
    if type is dispatch:
        require prepared and dispatches = 0
        require execution_authorized and not delivery_allowed                Dispatch only
        dispatches ← 1; last_exact ← true
require prepared and dispatches = 1 and last_exact                           Reject replay
return eligibility for the exact Core-prepared action                        Exact permit
```

Notation: previous is the prior event hash; sequence is the prior event number; prepared records one matched preparation; dispatches counts matched dispatches; last_exact marks the last durable record. The initial hash is 64 ASCII zero characters.

Execution contract: Control: only a unique current exact-action record can qualify dispatch; it never grants delivery. Source: runtime dispatch_proof::verify and daemon Journal. Hash-chain integrity assumes a trusted host; unknown effects never imply replay.

Unique current durable dispatch eligibility (A04). Exact identity and a unique current durable event constrain permit reuse. The algorithm implements dispatch qualification, not an automatic replay or delivery grant.

The conjunction prevents an older or duplicated journal record from qualifying the same prepared action through this permit path. It does not constitute a distributed exactly-once theorem: the journal is not an external trust root, and a process failure can leave the actual outcome unknown. The host's response to that uncertainty is to fence the attempted identity, preserve evidence for reconciliation, and avoid automatic native redispatch. A timeout is therefore not interpreted as proof that no effect occurred; crash-consistent update protocols require explicit assumptions about what survived persistence [33].

Thus uncertainty restricts permission instead of enlarging it. In a long trajectory, a reporting failure cannot silently create a second opportunity to perform the same consequential action. This is the recovery counterpart of external acceptance authority: an incomplete observation is a reason to reconcile the old transition, not evidence authorizing a new one.

3.6 Trusted evidence admission and obligation closure

Once an authorized action has executed, the next question is which observations may alter its acceptance state. Figure 6 separates production of a trusted receipt from admission of that receipt. The native bridge requires a settled matching run, independently reruns the registered validator through the configured sandbox, verifies the command pin, remeasures the workspace, and re-reads captured artifact bytes. Capture files live outside the candidate worktree so evidence collection cannot change the measured source. A model-supplied receipt digest or PASS field is not accepted as this execution.

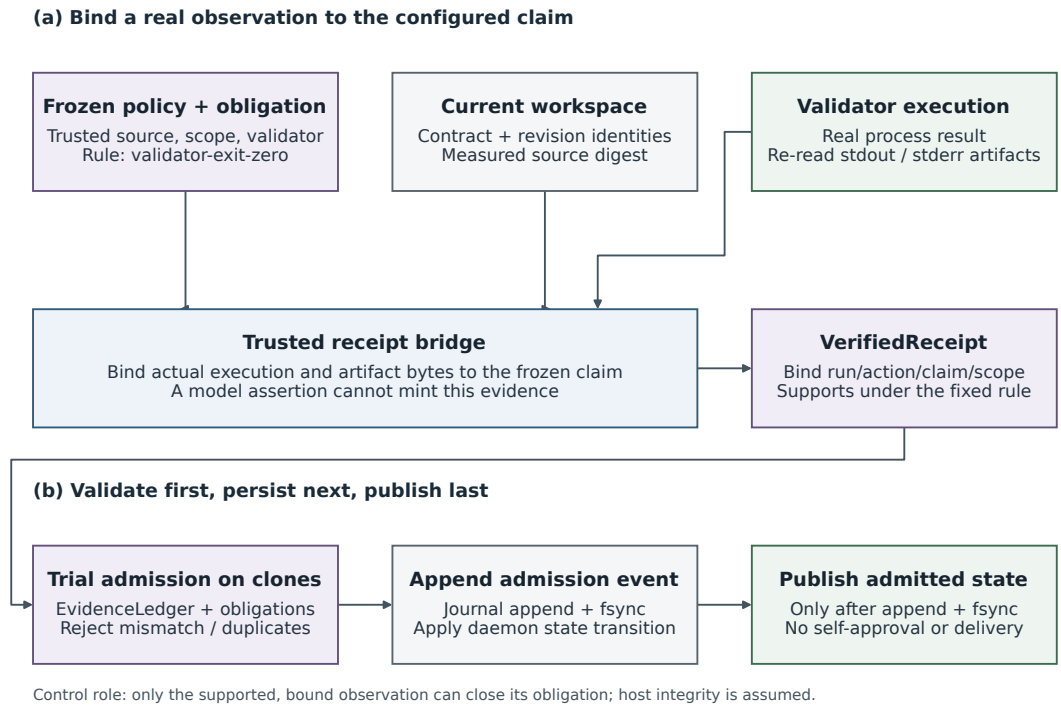


Figure 6: Trusted observation and durable evidence admission. Real validator execution and artifact checks precede the typed receipt. Private trial state becomes published evidence only after durable admission. This blocks self-issued success under the supported obligation; it does not turn validator exit zero into a universal safety claim.

In A05, e denotes evidence with its trusted typed receipt. Admission requires

$$G_{\text{admit}}(e) = I_e \wedge C_e \wedge S_e \wedge T_e \wedge V_e \wedge D_e. \tag{3.9}$$

I_e binds action, observation, obligation, claim, and receipt identities; C_e binds the current frozen contract and workspace; S_e checks trusted source and contained scope. T_e requires completed,

integrity-verified, rule-verified observation with the expected type and admission rule; V_e checks pinned validator identity/version and requirement coverage; D_e checks unused evidence, observation, and receipt identities within capacity. Scope containment is the production repository relation, including path-component boundaries, rather than a raw string-prefix comparison.

The bridge also persists and re-reads a transcript under the separate `voder-vgr5-trusted-validator-v1` domain. It binds session/run/action, contract/workspace, the parent action receipt, obligation/observation, validator identity/version, trusted source, and execution/output digests. This validation identity is distinct from the final-commit identity derived in Section 3.7.

Algorithm 2 Trusted evidence admission and obligation closure (A05)

Inputs: Frozen contract, current workspace, ledger L, obligations O, typed receipt e.

1:	**require** settled native observation bound to the exact action	
2:	execute the pinned validator through the trusted bridge	Run validator
3:	**require** completion, command identity and real artifact hashes	
4:	construct the sealed Core receipt for the supported rule	Typed receipt
5:	$L^* \leftarrow \text{clone}(L)$; $O^* \leftarrow \text{clone}(O)$	Private trial
6:	**require** exact run/action/observation/obligation/claim binding	Bind evidence
7:	**require** current contract/workspace, trusted source and scope	
8:	**require** validator version and requirement coverage	
9:	**require** rule/type match and unused evidence/receipt identities	Admission gate
10:	admit evidence on Ltrial	
11:	**if** target is Blocked: preserve Blocked	
12:	**else if** target is not Closed and closure relation matches:	
13:	close only if all effective dependencies are Closed	Dependencies
14:	append evidence_admitted event and fsync the journal	Persist first
15:	**if** append or fsync fails: STOP; publish neither trial object	Fail closed
16:	apply the durable daemon state transition	
17:	$L \leftarrow L^*$; $O \leftarrow O^*$	Publish
18:	**return** admitted evidence; delivery is still disallowed	Not delivery

Notation: L and O are the published evidence ledger and obligation registry; e is the trusted typed receipt. Ltrial and Otrial are private copies, displayed as L* and O*. Closure uses effective current-workspace obligation states.

Execution contract: Control: generated claims cannot publish acceptance state without trusted admission and durability. Source: trusted receipt bridge, Core admission and daemon finalization. Concrete gateway rule: pinned validator exit zero; one configured success obligation.

Trusted evidence admission (A05). A generated claim cannot publish acceptance state without the bound trusted observation and durable admission. The executable steps are unchanged; the control role is explicit in the execution contract.

Evidence admission and obligation closure are related but distinct. For target obligation q, required closure relation ρ_q, receipt relation $\rho(e)$, and prerequisite lookup deps, the closure condition is

$$\begin{aligned} \text{Close}(q, e) = {} & [\text{status}(q) \neq \text{Blocked}] \\ & \wedge [\text{status}(q) \neq \text{Closed}] \wedge [\rho(e) = \rho_q] \\ & \wedge \bigwedge_{d \in \text{deps}(q)} [\text{status}(d) = \text{Closed}]. \end{aligned} \tag{3.10}$$

d indexes a prerequisite, and status means effective status under the current contract/workspace. A blocked obligation remains blocked; unrelated or differently related evidence cannot silently close it. The concrete bridge supports `validator-process-exit` under `voder-validator-exit-zero-v1`, issuing a Supports relation. The gateway currently instantiates one success obligation. Generic relation and dependency support in Core does not imply a deployed interpreter for arbitrary natural-language requirements.

The decisive publication order is

$$\begin{aligned} (L^*, O^*) \leftarrow {} & \text{TrialAdmit}(\text{clone}(L), \text{clone}(O), e); \\ & \text{Persist}(\texttt{evidence_admitted}); \\ (L, O) \leftarrow {} & (L^*, O^*). \end{aligned} \tag{3.11}$$

L and O are the published evidence ledger and obligation registry; L^* and O^* are private trial replacements. TrialAdmit performs Core validation and evidence application on those copies. Algorithm 2

makes the persistence boundary explicit: validation failure leaves published objects unchanged, while append/fsync failure stops the writer instead of publishing an undurable trial state. A partial persistence failure requires recovery inspection; it is not permission to continue appending.

This gate addresses self-issued success directly. The proposer may supply a claim and suggest a repair, but it cannot turn that claim into admitted native evidence merely by repeating it, recalling an earlier PASS, or choosing a plausible digest. The trust boundary authenticates the specified observation; extending the observation to richer safety requirements remains a specification and verifier task [10, 16].

3.7 Fresh meta-audit and independent commit confirmation

An admitted receipt records an accepted observation, but final commitment requires the hard conditions to remain valid at closure. A06 therefore rechecks native context and obligation prerequisites, executes the validators bound to hard requirements, and requires an unchanged measured workspace. Let $\mathcal{R}_{\mathrm{h}}$ be the hard-requirement set, $\mathcal{V}(r)$ its frozen validator bindings for requirement r, and v one such validator. The meta-audit gate is

$$\begin{aligned} G_{\text{audit}} = G_{\text{context}} \wedge G_{\text{hard}} \\ \wedge \Big(\bigwedge_{r \in \mathcal{R}_{\mathrm{h}}} \bigwedge_{v \in \mathcal{V}(r)} \text{FreshPass}(v) \Big) \\ \wedge [W_{\text{after}} = W_{\text{before}}]. \end{aligned} \tag{3.12}$$

G_{context} checks exact stage/action/contract/workspace identity; G_{hard} checks hard-evidence and closure prerequisites. FreshPass is a new pinned execution with artifact checks, not a recalled PASS. Absent validator bindings cannot satisfy the conjunction vacuously. Executions are deduplicated by validator ID, an empty outcome list is rejected, and W_{before} and W_{after} are measured workspace identities around the audit.

A successful audit has its own durable digest. Final commitment then binds session s, run u, action a, contract digest d_{C}, workspace digest d_{W}, and audit digest d_{M}:

$$\begin{aligned} d_{\mathrm{c}} = H\Big(\text{UTF8}\big(\texttt{voder-vgr4-commit-v1} \| \texttt{\textbackslash 0} \\ \| s \| \texttt{\textbackslash 0} \| u \| \texttt{\textbackslash 0} \| a \| \texttt{\textbackslash 0} \\ \| d_{\mathrm{C}} \| \texttt{\textbackslash 0} \| d_{\mathrm{W}} \| \texttt{\textbackslash 0} \| d_{\mathrm{M}} \big) \Big). \end{aligned} \tag{3.13}$$

d_{c} is the final-commit digest and `\0` denotes a NUL separator; display wrapping introduces no bytes. The `vgr4` domain text is an implementation identifier, not a product-version claim. The daemon persists final commitment as a separate fsynced event, reconstructs state again, and checks the final identity and candidate bytes before returning a committed receipt.

The broker still does not trust that producer response as sufficient authority. After the writer exits, `NativeVgrCommitVerifier` starts a read-only Rust observer, resumes the expected session, resolves one matching run, and requires complete bounded replay. For replayed event sequence F and member f, confirmation requires

$$\begin{aligned} G_{\text{confirm}} = [\text{RunState} = \text{VGR_COMMITTED}] \\ \wedge [\text{DeliveryAllowed} = \text{true}] \\ \wedge [\text{StateDigest} = d_{\mathrm{c}}] \wedge \text{ReplayComplete} \\ \wedge \big| \{ f \in F : \text{type}(f) = \texttt{vgr.final_committed} \} \big| = 1 \\ \wedge \text{ExactFinalIdentity}\,. \end{aligned} \tag{3.14}$$

ExactFinalIdentity means that the unique event's session, run, action, and commit digest equal the expected request and receipt. Algorithm 3 includes writer-lock release and observer cleanup: confirmation is not a concurrent read of a still-mutating producer journal.

Algorithm 3 Meta-audit and independent final-commit confirmation (A06)

Inputs: Exact action, frozen contract, admitted hard evidence and candidate workspace.

```
require current context is VGR_EVIDENCE_ADMITTED
require hard evidence and closure prerequisites
validators ← unique IDs bound to hard requirements                    Deduplicate
require validators is not empty
for each validator in validators:
    rerun the pinned validator and verify its real outputs            Reverify
require workspace identity is unchanged                               Unchanged tree
persist VGR_META_AUDITED and its audit digest                         Durable audit
recheck context/workspace; compute the final-commit digest
append final_committed as a separate fsynced event                    Final commit
release the producer writer lock                                      Release lock
start an independent read-only Rust observer                          Independent
resume the expected session and resolve exactly one run
require VGR_COMMITTED, delivery_allowed and matching digest
replay the run events within the bounded verification window
require replay is complete                                            Complete replay
require exactly one final_committed event                             Unique event
require its session/run/action/digest exactly match                   Bind identity
wait for observer cleanup and lock release                            Cleanup
only then report confirmed native commitment to the broker            Confirm
```

Notation: validators is the unique set of pinned validator IDs covering hard requirements. The final digest binds session, run, action, contract, workspace and audit identities. VGR_COMMITTED is a native qualification, separate from user-workspace delivery.

Execution contract: Control: the producer report cannot self-confirm; fresh audit and independent replay must agree. Source: native_closure, NativeVgrCommitVerifier and VgrBroker. Native commitment precedes the separate workspace-delivery transaction.

Fresh meta-audit and independent confirmation (A06). The producer cannot confirm its own success by returning a receipt. Current checks, reconstructed state, and one exact final event must agree before the broker proceeds.

Consequently, native commitment can remain valid even if the subsequent host checkpoint fails. The broker records `VERIFIED_NOT_DELIVERED`, preserves confirmed native authority, and stops before delivery. This distinction prevents a reporting fault from being treated either as an undo of durable commitment or as justification to execute the action again. Native `COMMITTED`, Git commit, and completed workspace delivery are three different facts.

Independent confirmation separates the actor that produces a success report from the procedure that establishes its authoritative meaning. Fresh checks guard against stale acceptance conditions, and replay establishes the exact durable outcome. Independence here is procedural and process-level under the trusted host; it does not imply a separate trust root or an evaluator capable of judging every future ASI behavior.

3.8 Verified delivery and conservative restoration

Preserving native commitment through reporting faults leaves one remaining obligation: transferring only the verified candidate bytes into the live workspace. A07 implements this as a second transaction, shown in Figure 7, with its own exact operator policy, candidate-worktree identity, target locks, before images, and delivery intent. For target index i among n approved members, let w_i be live bytes, v_i verified candidate bytes, and n_i the expected final digest. Using r_i, o_i, x_i from Section 3.3, the byte conditions are

$$H(w_i) = o_i \wedge v_i = \mathrm{UTF8}(x_i), \qquad n_i = H(v_i). \tag{3.15}$$

After policy, authority, path-safety, and lock checks, the host durably stores before images and a fresh delivery intent. Each replacement rechecks preimage and race-sensitive file identity, writes and fsyncs a fresh temporary file, renames it into place, and syncs the parent directory. Final hashes are re-read before persisting `DELIVERED`. Single-file and batch implementations differ in the placement of preliminary policy checks, while retaining these lock-protected byte conditions.

Recovery must preserve intervening user work rather than merely make the transaction appear rolled back. Let w_i^{now} be bytes inspected during recovery. Restoration is eligible only when every member

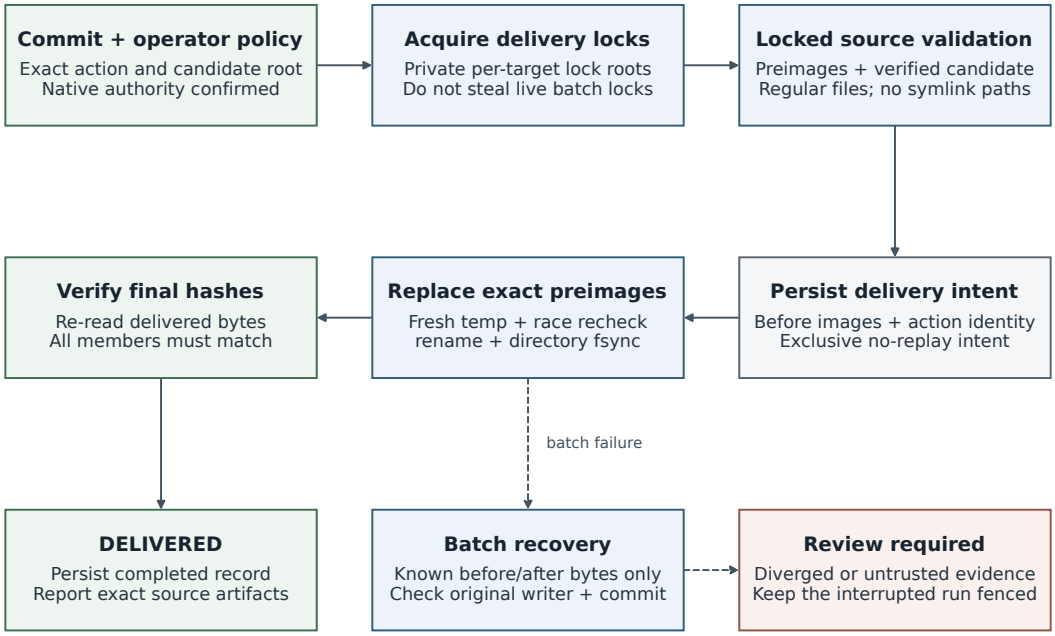


Figure 7: Authorization carried through to destination bytes. Independent native confirmation precedes preimage, candidate, locking, and final-hash checks. Known-state restoration protects intervening work; ambiguous state requires reconciliation. The control property concerns qualified writeback, not reversibility of arbitrary external effects.

is in a known before or after state:

$$
\begin{aligned}
G_{\text{restore}} &= \bigwedge_{i=1}^{n} [H(w_i^{\text{now}}) \in \{o_i, n_i\}], \\
\mathcal{P}_{\text{restore}} &= \text{reverse}([r_i : H(w_i^{\text{now}}) \neq o_i]).
\end{aligned}
\tag{3.16}
$$

$\mathcal{P}_{\text{restore}}$ is the reverse-order sequence of paths needing restoration. All members are inspected before restoration starts; unknown or divergent bytes require operator review. Reconciliation additionally checks original-writer liveness, exclusive recovery ownership, and independently confirmed native authority. It never re-dispatches the native action to repair host writeback.

The guarantee is recoverable coordination under the host's locks and filesystem assumptions. Sequential renames do not create instantaneous cross-file visibility for arbitrary external readers, and successful postchecks do not prevent later edits. The final delivery record describes the verified transfer that completed, not permanent validity of the workspace.

This delivery boundary completes the implemented response to the control problem: an accepted verdict must govern the actual destination transition. Conservative restoration also protects state created outside the failed transaction; divergent bytes are not erased simply to manufacture a clean rollback. These are local software protections on which broader controlled-action systems could build, subject to effect-specific protocols.

3.9 Final-source visual acceptance

For visual creation, byte-correct transfer is useful only if the reviewed rendering belongs to the bytes being delivered. A08 joins visual evidence and candidate verification by source identity, rather than by conversational proximity. Figure 8 shows independent render and verification paths: HTML uses real browser inspection, SVG uses source-bound rendering, and actual exposed frames are recorded before a model decision can contribute to readiness.

Algorithm 4 Verified writeback and conservative recovery (A07)

Inputs: Independently confirmed commit, exact policy and verified candidate bytes.

1:	acquire private target locks for the exact operator scope	Lock targets
2:	validate policy, safe paths and the candidate-worktree identity	
3:	**for** every member: check live preimage and verified replacement	Check preimage
4:	persist before images and a fresh delivery intent	Durable intent
5:	**for** every target member:	
6:	recheck its expected bytes and race-sensitive file identity	
7:	create and fsync a fresh temporary file	Stage bytes
8:	rename into place and fsync the parent directory	Replace + sync
9:	re-read all expected final hashes	
10:	persist DELIVERED and report the exact source artifacts	Deliver
11:	**if** batch writeback fails:	
12:	inspect all members before attempting restoration	Inspect all
13:	**require** each current digest is a known before/after digest	Known states
14:	restore changed members in reverse order	Restore reverse
15:	recheck restored hashes and persist rollback record	
16:	otherwise keep the run fenced for operator review	Fence
17:	operator reconciliation also checks original writer liveness	Writer liveness
18:	never re-dispatch the native action to recover delivery	No re-execution

Notation: A preimage is the exact expected source byte digest. Before/after digests identify original and verified replacement bytes for each target. A fresh intent binds these values to the independently confirmed action and its candidate worktree.

Execution contract: Control: accepted authority is bound to delivered bytes; divergence fences restoration. Source: VerifiedWorkspaceDelivery / BatchWorkspaceDelivery. Batch coordination is not instantaneous multi-file visibility to external readers.

Verified writeback and conservative recovery (A07). Accepted authority remains bound to target preimages and final bytes. Divergence fences restoration, and writeback recovery cannot redispatch the native action.

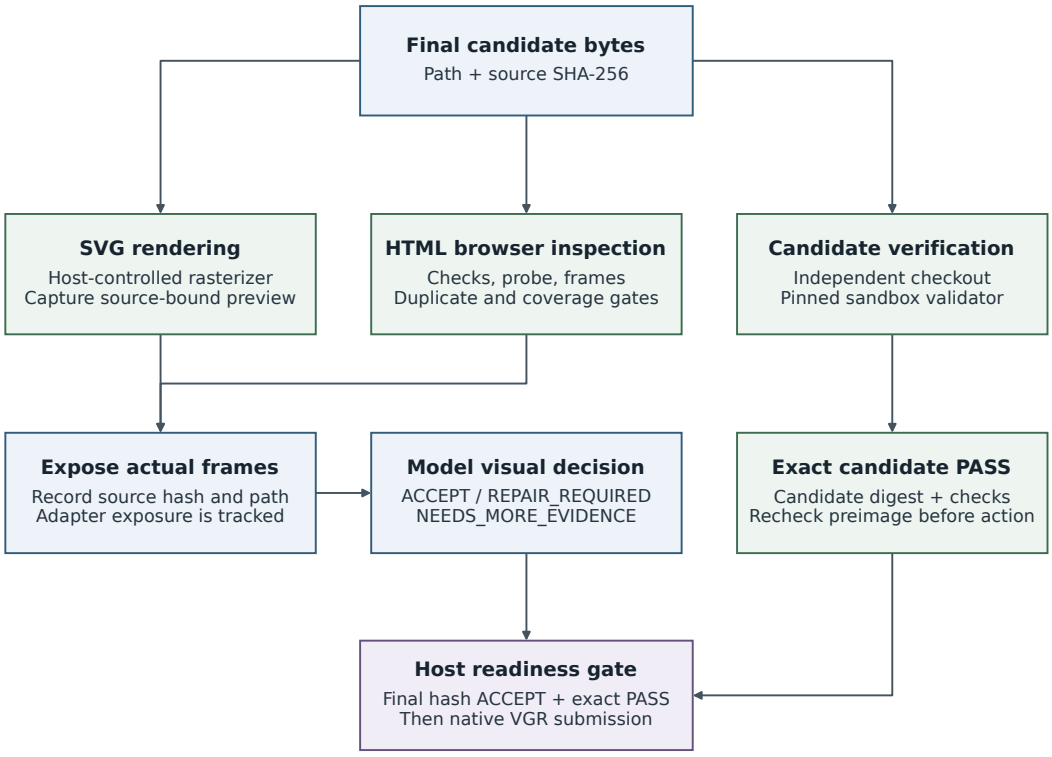


Figure 8: Final-source visual evidence and submission readiness. Render identity, frame exposure, and model ACCEPT refer to the final source path and hash. Exact candidate PASS is a separate prerequisite; neither browser success nor visual review grants native authority.

For candidate sequence P and relative-path accessor $r(p)$, the visual subset is

$$\mathcal{P}_{\mathrm{v}} = \{p \in P : r(p) \text{ ends in case-sensitive } \texttt{.html} \text{ or } \texttt{.svg}\}. \tag{3.17}$$

The suffix test is case-sensitive. With predicates defined below, readiness is

$$\begin{aligned} \mathrm{VisualOK}(p) = {} & \mathrm{CurrentRendered}(p) \wedge \mathrm{Exposed}(p) \\ & \wedge [\mathrm{Decision}(p) = \mathrm{ACCEPT}] \wedge \mathrm{CorrectKind}(p), \\ \mathrm{Readiness}(P) = {} & \begin{cases} \mathrm{WAIT}, & \text{invalid shape or scope mismatch;} \\ \mathrm{EXPLICIT_ONLY}, & \mathcal{P}_{\mathrm{v}} = \emptyset; \\ \mathrm{READY}, & \forall p \in \mathcal{P}_{\mathrm{v}},\ \mathrm{VisualOK}(p); \\ \mathrm{WAIT}, & \text{otherwise.} \end{cases} \end{aligned} \tag{3.18}$$

CurrentRendered requires the latest status `RENDERED` and exact source path/hash; Exposed records actual frame exposure. Decision is the recorded model decision, and CorrectKind requires `HTML_BROWSER` for HTML and the source-bound rendering path for SVG. `READY` additionally requires all visual members to satisfy the conjunction. A nonvisual proposal remains `EXPLICIT_ONLY`, requiring explicit verified submission rather than automatic visual finalization.

Host submission further requires authenticated PASS for the exact candidate, current source preimages, matching scope, and no consumed, in-flight, or uncertain native commitment. Visual evidence may precede or follow a candidate check; no fixed render-before-first-check order is assumed. An exposed but unaccepted exact rendering can block repeated verification. `REPAIR_REQUIRED` requires changed source, while `NEEDS_MORE_EVIDENCE` requires specific uncovered requirements and a useful different probe. These rules prevent old frames from accepting new bytes; they do not make model aesthetic judgment an independent certificate of quality.

3.10 Bounded memory with preserved provenance

The same identity discipline must survive context changes without turning remembered observations into renewed authority. Figure 9 separates project records, segmented complete dialogue pairs, the existing index, policy selection, and bounded composition. Task settlement records completed or interrupted work; the recall tool exposes `recent/search/read`, validates archive parts and physical source lines, and pages long text. These mechanisms restore inspectable history rather than silently treating a generated summary as the original record.

A09 uses explicit lexical ranking. For current goal g, memory row m, and source token-set function T,

$$\begin{aligned} \mathrm{score}(g, m) = \big| T(g) \cap T(m.\mathrm{goal} \| \mathrm{space} \| m.\mathrm{answer} \\ \| \mathrm{space} \| m.\mathrm{content}) \big|. \end{aligned} \tag{3.19}$$

T lowercases text, takes the first 100 matched ASCII word/hyphen tokens of length at least three, and adds Chinese bigrams from at most 160 starting positions in each matched CJK segment. Here space is one separating space; absent text fields contribute no tokens. This is implemented token overlap, not an embedding similarity model.

Selection applies the effective saved-plus-current-turn policy p before ranking:

$$\begin{aligned} \mathrm{Allow}(m, p) = {} & [p.\mathrm{mode} \neq \mathrm{off}] \wedge \neg \mathrm{Excluded}(m, p) \\ & \wedge [p.\mathrm{mode} \neq \mathrm{only} \vee \mathrm{Included}(m, p)]. \end{aligned} \tag{3.20}$$

p.mode is `auto`, `off`, or `only`; Excluded, Included, and Prefer are the source's exact-phrase policy predicates. Ordinary historical rows sort by descending $(\mathrm{Prefer}(m, p), \mathrm{score}(g, m), i)$, where i is original row position; explicit recent recall instead uses $(\mathrm{Prefer}(m, p), i)$. Topic selection requires

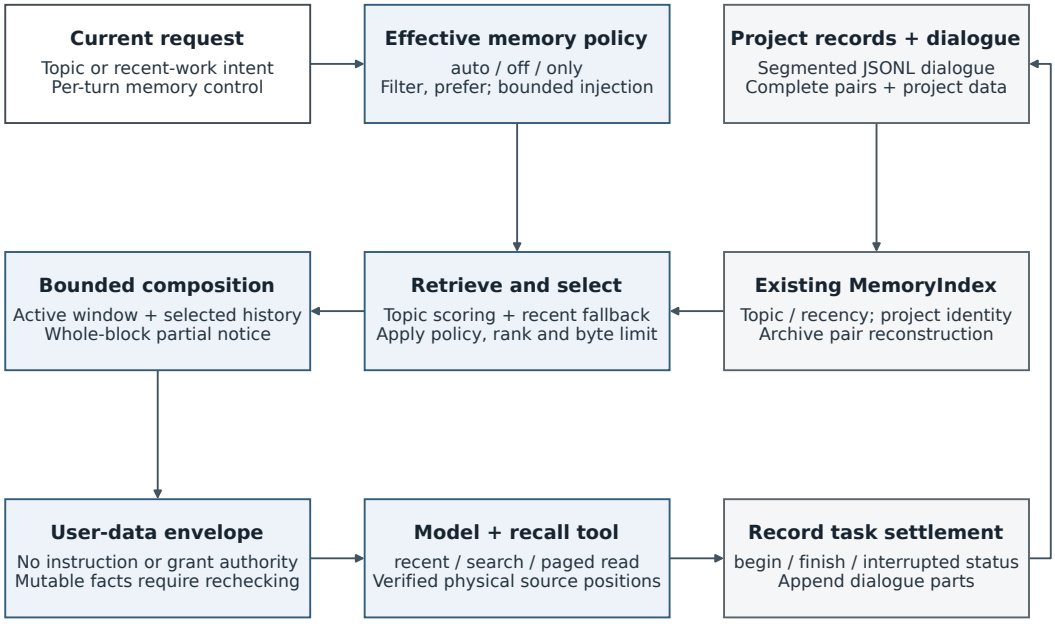


Figure 9: Project memory, bounded recall, and dialogue provenance. Policy governs selection; archive and source-line checks govern recall provenance. Context remains untrusted user data and cannot restore an old write permission or validation grant.

positive overlap or preference, while recent recall admits recency candidates. Selection is capped at $\min(15, p.\text{limit})$ and then restored to the prescribed display order.

The composer also constrains how selected history enters inference. For input block sequence $\mathcal{B}$, selected ordered subsequence S, and byte budget B,

$$|\operatorname{join}_2(S)|_8 \le B, \qquad S \subseteq_{\text{ordered}} \mathcal{B}. \tag{3.21}$$

join_2 inserts two newline characters between retained blocks. The base composer prioritizes active chat within $\min(17000, B - 500)$ bytes and fits other history within the remaining allowance. The dialogue enhancement admits at most $\min(6, p.\text{limit})$ archived pairs; the final host boundary independently fits complete blocks within 24576 bytes and explicitly marks partial context. Greedy priority order is not a global knapsack optimum. Base summaries may contain bounded excerpts, whereas archive-only dialogue preserves complete stored pairs or reports omission. All injected history remains user data: mutable facts require rechecking, and recalled receipts cannot authorize current execution.

The cost consequence of A09 is selective inference input, not deletion of engineering history. Archival volume can grow without requiring every retained pair to be sent on each request; only policy-selected blocks enter the bounded supplement. Selection uses local lexical operations rather than a mandatory model call to summarize the entire archive. This reduces the history payload relative to full replay whenever the retained archive exceeds the selected subset, without asserting a byte-to-token conversion or a provider-specific discount.

3.11 Read-only capabilities without authority escalation

Restored context can identify an observation not yet obtained, but obtaining that observation must preserve the native authority boundary. Figure 10 distinguishes three implemented seams. `InstalledSkills` selects SHA-pinned task material within a budget and rehashes selected records; the ordinary VS Code constructor does not currently supply a `skillStore`. `HostToolRegistry`

accepts host-built read/search/verify adapters. `EvidenceSearchService` routes repository, local-wiki, and configured public retrieval, returning source identities and untrusted findings. These are context and observation services, not alternative commitment paths.

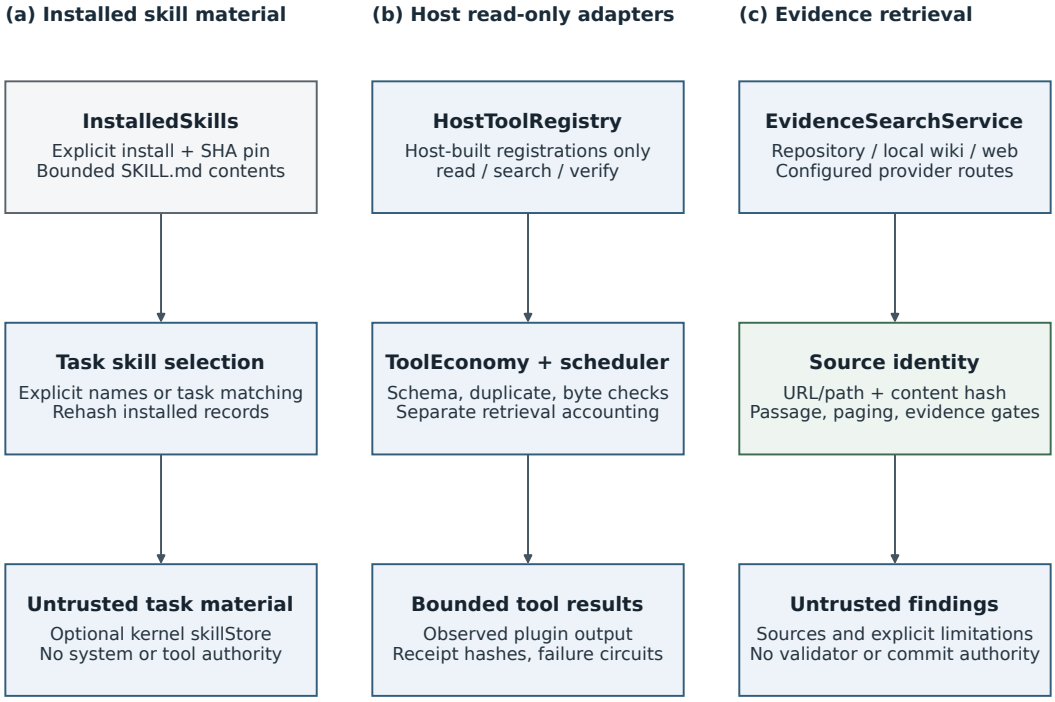


Figure 10: Knowledge and read-only capability seams. Skills, installed adapters, and retrieval have different admission and budgeting rules. The diagram distinguishes kernel support from ordinary-entry configuration; output hashes are not native validator evidence.

A10 guards read-only adapter use before its result reaches the model. Let t be the tool name, a validated arguments, and b its host-assigned budget class. Define the retrieval exception, exact input signature σ, and tool-economy gate by

$$\begin{aligned}
\text{Retrieval}(t,a,b) &= [b = \texttt{retrieval}] \\
&\quad \wedge [t = \texttt{voder_ext_search_evidence}] \\
&\quad \wedge [a.\text{mode} \in \{\texttt{web}, \texttt{research}, \texttt{fetch}, \texttt{read}\}], \\
\sigma &= H(\text{UTF8}(\text{JS}([t,a]))), \\
G_{\text{tool}} &= [f_t < F_{\max}] \wedge [\text{Retrieval}(t,a,b) \vee c_{\text{s}} < C_{\text{s}}] \\
&\quad \wedge [\sigma \notin D_{\text{f}} \cup D_{\text{i}}] \\
&\quad \wedge [\sigma \notin D_{\text{c}} \vee (\text{Retrieval}(t,a,b) \wedge a.\text{mode} = \texttt{fetch})].
\end{aligned} \tag{3.22}$$

f_t and $F_{\max}$ are the adapter failure count and circuit threshold; c_{s} and C_{s} are used and allowed shared extension calls. D_{f}, D_{i}, and D_{c} contain recent failed, in-flight, and completed signatures. A completed fetch may be retried through the stated retrieval exception; failed or in-flight duplicates remain refused. Separate production checks require a current session, a closed bounded scalar schema, serializable bounded output, and no cancellation. Read-adapter timeout cancellation does not cancel native actions or grant replay.

A10 therefore rejects a covered duplicate before executing its adapter, avoiding another lookup and another returned payload. Its recent-signature sets are bounded, and the completed-fetch exception is intentional: the mechanism suppresses specific redundant work rather than promising permanent

deduplication of all retrieval. Failure thresholds constrain fruitless adapter retries; they do not justify weakening a validator or treating unavailable information as success.

After removing entries above the configured risk ceiling, advisory capability ranking uses

$$s_t = 40\mathbf{1}[\text{cap}(t) = \text{need}(q)] + 10e_t - \tau_t/100 - \ell_t/200 - 15r_t. \tag{3.23}$$

s_t is score; e_t, τ_t, ℓ_t, and r_t are evidence-value, token, latency-in-milliseconds, and policy-risk hints. cap is the entry capability and need(q) maps planning purpose q to a capability: inspect/explain to read, locate to search, and validate to verify. Entries sort by descending score then name. This ranking neither optimizes global task success nor authorizes a side effect. Installed adapter code is trusted executable host code; labelling its returned data untrusted does not make arbitrary adapter installation safe.

3.12 Non-authoritative library extensions

The implemented authority path must remain distinct from supplementary algorithms in the same archive. Figure 11 records the boundary for choice and method evaluation; A11–A13 below are existing library mechanisms, not deployed stages of the ordinary gateway. Their inclusion completes the mathematical inventory without attributing unintegrated behavior to the production harness. Static call-site inspection found test use rather than ordinary production calls for the named APIs.

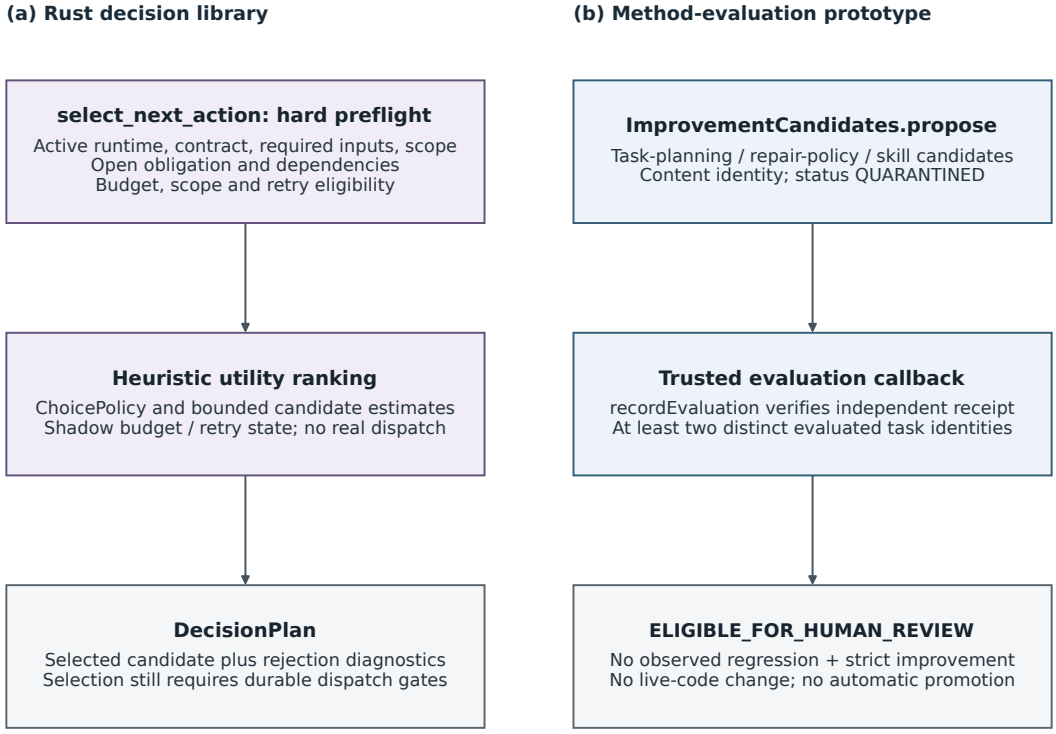


Figure 11: Library mechanisms outside ordinary dispatch. Heuristic choice returns a DecisionPlan, not a permit. Method evaluation can qualify a quarantined candidate for human review, not autonomously activate it. Neither lane is part of the current native authorization path.

Semantic retry eligibility (A11). The Core library excludes arbitrary action IDs from semantic retry identity. For UTF-8 byte-length-prefixed encoding LP and action signature α,

$$\begin{aligned} \mathrm{LP}(s) &= \mathrm{dec}(|s|_8) \| : \| s, \\ \alpha &= \texttt{voder-action-v1} \,|\, \| \,\mathrm{LP}(k) \| \,\mathrm{LP}(q) \| \,\mathrm{LP}(h) \\ &\quad \| \,\mathrm{LP}(t) \| \,\mathrm{LP}(d_{\mathrm{p}}) \| \,\mathrm{LP}(s). \end{aligned} \tag{3.24}$$

The fields $k, q, h, t, d_{\mathrm{p}}, s$ are action kind, optional target obligation, optional hypothesis, trusted tool ID, trusted canonical-parameter key, and validated repository scope. Absent fields encode as empty strings. This representation is length-prefixed, not hashed. Relevant already-admitted evidence is reduced to

$$\begin{aligned} E = \mathrm{sortunique}\{(e.\mathrm{claim}, e.\mathrm{relation}, e.\mathrm{contentDigest}) : e \in L, \\ e.\mathrm{obligation} = q,\ e.\mathrm{contractDigest} = d_{\mathrm{C}}, \\ e.\mathrm{workspaceDigest} = d_{\mathrm{W}},\ e.\mathrm{environment} = \varepsilon\}. \end{aligned} \tag{3.25}$$

Here L is the admitted fact ledger, e one fact, E its sorted duplicate-free semantic key, d_{C} the frozen contract digest, d_{W} the current workspace digest, and ε the contract environment. Let C denote the full immutable contract identity and N count prior records equal in $(\alpha, C, d_{\mathrm{W}}, E)$. An ordinary retry requires $N = 0$. Pre-authorized sampling additionally requires a matching frozen rule, ordinal $1 \le j \le J$, no matching ordinary attempt, and $j = N + 1$, with J the rule's maximum sample count. Renaming the action or changing a timestamp cannot refresh unchanged semantic state. No production gateway caller of `RetryLedger::try_record` was found; the concrete host fences in Section 3.5 must not be confused with this library rule.

Hard-filtered heuristic choice (A12). The separate choice library considers at most 128 candidate actions. With normalized integer hints in $[0, 100]$, its default utility is

$$U(a) = 8c_a + 6i_a + 4p_a + 3v_a - 4t_a - 2\ell_a - 6r_a - 2x_a. \tag{3.26}$$

For action a, c_a, i_a, p_a, v_a estimate closure, information, progress, and reversibility; t_a, ℓ_a, r_a, x_a estimate token, latency, risk, and context cost. These are heuristic hints, not probabilities or verified outcomes. Eligibility precedes ranking:

$$\begin{aligned} \mathcal{A}_{\text{eligible}} = \{a \in \mathcal{A} : G(a) \wedge (c_a > 0 \vee i_a > 0 \vee p_a > 0) \\ \wedge\, U(a) \ge U_{\min}\}. \end{aligned} \tag{3.27}$$

$\mathcal{A}$ is the bounded candidate sequence, $\mathcal{A}_{\text{eligible}}$ its eligible subset, and $U_{\min} \ge 0$ the controller's utility floor. G includes exact active context and unique identities, inputs, available reservation, budget feasibility, applicable open obligations and closed dependencies, plus preparation and retry checks on private copies. The maximum utility wins, with ascending action ID breaking ties. Frozen weights may replace the defaults; valid weights are integers in $[0, 1000]$, not all zero. No rejected candidate mutates live budget or retry state, and the returned `DecisionPlan` remains advisory.

Quarantined method evaluation (A13). The diagnostics prototype likewise restricts what favorable observations can authorize. For n distinct verified task evaluations indexed by j, let b_j, c_j be baseline/candidate pass flags, g_j an independently verified regression-free flag, and u_j, v_j their non-negative finite costs in a common evaluator unit. Then

$$\begin{aligned} S &= \bigwedge_{j=1}^{n} [g_j \wedge (\neg b_j \vee c_j)], \\ I &= \bigvee_{j=1}^{n} [(\neg b_j \wedge c_j) \vee (b_j \wedge c_j \wedge v_j < u_j)], \\ \mathrm{Eligible} &= [n \ge 2] \wedge S \wedge I. \end{aligned} \tag{3.28}$$

S denotes no observed regression and I at least one strict observed improvement. A host-configured `verifyEvaluation` callback must validate each claim and receipt digest; duplicate task IDs are rejected. Eligibility means `ELIGIBLE_FOR_HUMAN_REVIEW`; otherwise the method stays quarantined. No production importer or autonomous activation caller was found. These observations neither establish universal absence of regressions nor implement a deployed self-improvement lifecycle.

3.13 Composition and guarantee boundary

The core novelty lies in how the checks compose, rather than in any single digest or validator. To state the resulting property precisely, let K be an ordinary evidence-mode delivery request for candidate P, CPass(P) its authenticated candidate PASS, Pinned(K) the exact operator policy and candidate-root binding, and Bytes(K) the locked preimage, candidate-byte, and final-hash checks. Let DurableDelivery(K) mean that the normal path completed its delivery postchecks and persisted `DELIVERED`. Under the trusted execution and persistence assumptions above, the implemented path enforces

$$\begin{aligned}\text{DurableDelivery}(K) \implies &\text{CPass}(P) \wedge \text{Pinned}(K) \\ &\wedge G_{\text{confirm}} \wedge \text{Bytes}(K).\end{aligned} \tag{3.29}$$

This implication refers to checks at their specified execution points, not to a single atomic observation of every file or to the workspace after later user edits. Native confirmation is itself downstream of the exact durable dispatch, trusted evidence admission, and unchanged-workspace meta-audit. The evidence at one stage therefore cannot substitute for the distinct authority required at the next.

The argument follows the only normal transition path that can issue the delivery record. The scheduler requires exact PASS and current source pins before ordinary submission. The broker requires independent native confirmation before calling delivery. The delivery implementation rejects mismatched policy, source preimages, or candidate bytes, and persists completion only after its final hash checks. A failed host checkpoint blocks delivery while retaining native commitment; interrupted delivery remains subject to known-state restoration or fenced reconciliation. Removing any required conjunct therefore prevents normal issuance of the successful record, rather than merely producing a warning that the model may ignore.

This is the Harness realization of E2's evidence-governed persistence principle: proposals can guide computation, but only accepted evidence at the correct identity and authority boundary can advance persistent task state. It is not a proof that the configured validator captures every requirement, that visual judgment is objective, that an attacker-controlled host is safe, or that the harness outperforms another system. The current single-obligation gateway, the library integration boundary, and the separation between native commitment and delivery define the precise scope of the mechanism.

The resulting control argument has two distinct layers. VGR enforces that the declared acceptance conditions precede successful delivery; the operator and verifier determine which substantive requirements those conditions express. The first layer is the implemented architectural contribution. Strengthening the second layer can extend the assurance carried by the same boundary, provided identity, authority, and mediation are preserved. This is the specific connection to safe ASI-agent exploration developed in Section 7.

4 Engineering Implementation

Voder's source code will be released publicly on GitHub. The implementation described here is the supplied source snapshot, rather than a claim that the repository, release automation, or installation artifacts have already been published. Its engineering objective is to preserve the authority separation established in Section 3.13 across concrete model sessions, processes, files, and editor interactions: a proposal may initiate computation, but each persistent transition must obtain the evidence required by its own owner.

The implementation translates Figure 2 into concrete ownership decisions: the session can request tools, the host fixes scope, the native path qualifies effects, and delivery checks the destination. Model upgrades, longer contexts, or a new editor frontend do not themselves change those owners. This is how the present software architecture investigates control as competence increases, without claiming an implemented ASI system.

Cost control follows the same boundary: eliminate redundant proposal, observation, and transport work while preserving the checks that authorize persistence. The implemented opportunities are local rejection before remote execution, exact-request evidence replay, bounded model input, and suppression of duplicate attempts. The following modules identify both the avoided operation and the condition that makes avoiding it valid; they do not assume that a cheaper proposal can replace independent native verification.

4.1 Implementation lineage and integration sequence

That separation determines Voder's relationship to its execution substrate. Voder develops the VeriLoop Harness control logic while adapting part of Pi's implementation for model interaction [17]. The attribution record identifies the initial Pi baseline as version 1.1.0, commit `abe508e1b899`; the supplied tree retains upstream source, copyright notices, and MIT licensing under the engine directory. The independently developed Rust VGR component supplies native decision and persistence authority. Consequently, neither the reused session machinery nor retained upstream package names identify the owner of Voder's acceptance policy.

The source supports a concrete integration sequence without requiring an invented development chronology. The early provenance document specifies delegation to Pi's public SDK and treats native integration as unfinished work. The current tree then supplies an explicit model host, a broker and task kernel, candidate verification, the Rust gateway and independent observer, verified delivery, and editor control modules. These artifacts establish the progression from a reused interaction substrate to an integrated evidence-governed execution path; they do not establish dates, authorship percentages, or a claim that Pi was wholly rewritten. Where the early handoff description differs from the current implementation, the executable path defines the present scope.

The resulting contribution is the composition of these layers. Pi provides reusable session execution, streaming, and context-compaction facilities [17]; Voder supplies the permitted tool surface, task identities, candidate admission, native dispatch, reconciliation, and delivery conditions around them. The VGR transfer from VeriLoop E2 therefore changes which subsystem may authorize persistence, rather than replacing an upstream agent loop with another loop that merely restates verification instructions.

4.2 Build roots and deployment contract

This division of responsibility is reflected in the build layout. Figure 12 separates the native workspace, gateway crate, product JavaScript modules, upstream engine workspace, and editor package. The root Cargo workspace contains `voder-core`, `voder-runtime`, `voder-protocol`, and `voderd`. The gateway has its own Cargo workspace and path dependencies on the native crates. In contrast, the root npm workspace includes only the provider package: its TypeScript build is not a build of the complete product.

For an engineering checkout, the manifest-defined native build targets can be made explicit:

```
cargo build --locked --release -p voderd \
  --target-dir .voder-build
cargo build --locked --release \
  --manifest-path voder-agent/crates/voder-gateway/Cargo.toml \
  --target-dir .voder-build
```

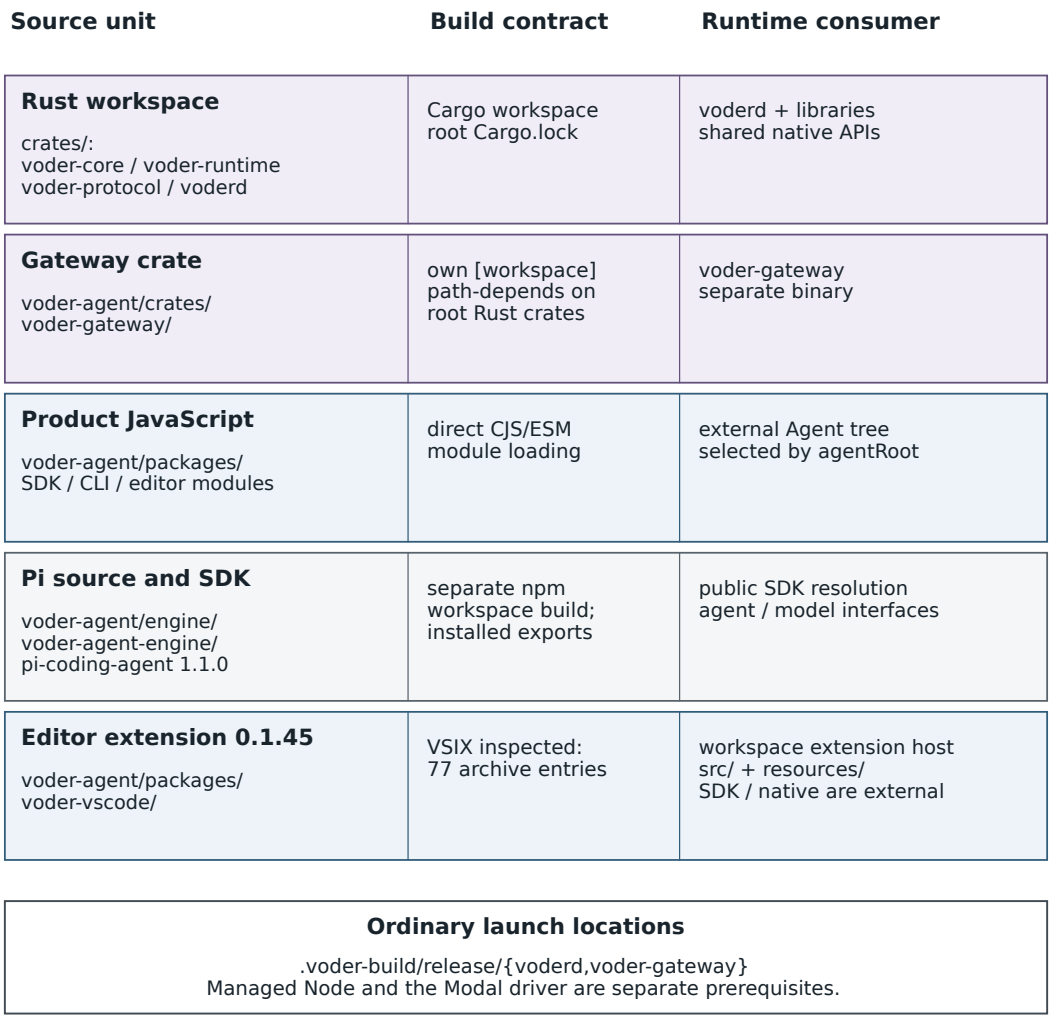


Figure 12: Build roots and deployable boundaries. The native workspace, separate gateway crate, product JavaScript modules, and Pi source workspace have distinct build or loading contracts. The inspected VSIX contains the editor adapter; the Agent SDK and native runtime are resolved separately. The ordinary launcher also identifies the managed Node executable and Modal driver explicitly.

These commands identify the two native build roots; they are not a complete installation recipe or evidence of a clean-machine build. Model-provider configuration, the Pi SDK dependencies, the managed Node runtime, and the Modal driver remain separate deployment prerequisites. Product JavaScript is loaded from its module tree, whereas the vendored Pi monorepo retains its own package build contract.

The inspected version 0.1.45 VSIX contains 77 archive entries and the editor adapter, not a bundled copy of every SDK module or native executable. The shipped user-facing integration is a VS Code extension: it supplies the task panel, operator approvals, selected-code and workspace context, and progress or result presentation, while the separate host process, Pi session, and Rust VGR binaries retain execution and authorization responsibilities. Thus VS Code is the current client surface, not an intrinsic dependency of the VGR acceptance rule. Separation of reusable agent runtime and client-specific surfaces is also documented for Codex [18]. Its extension runs in the workspace extension host, which is significant for remote development: process paths and permissions refer to that host. Ordinary launch resolves the agent root, native binaries, managed Node executable, and sandbox driver separately.

When the managed Node executable has a different owner, the launcher copies bytes from a no-follow file descriptor into a private content-addressed executable cache and checks source identity and digest, rather than changing permissions on the upstream installation. This explicit dependency boundary makes an absent component diagnosable, but also means that the current artifact should not be described as a self-contained, universally portable installer.

4.3 Pi adaptation and the model-session boundary

Once these components are located, `PiModelHost` composes the reused session machinery with a closed host-defined capability surface. It constructs a `DefaultResourceLoader` with implicit extensions, skills, prompt templates, and context files disabled, and supplies the Voder system-prompt override. `createAgentSession` receives `noTools: 'all'`, the explicitly approved tool names, and their custom implementations. The in-memory Pi session manager holds the live interaction; durable Voder control state is maintained separately. This separation responds to continuity issues documented in long-running coding-agent workflows, without importing their control algorithms [21].

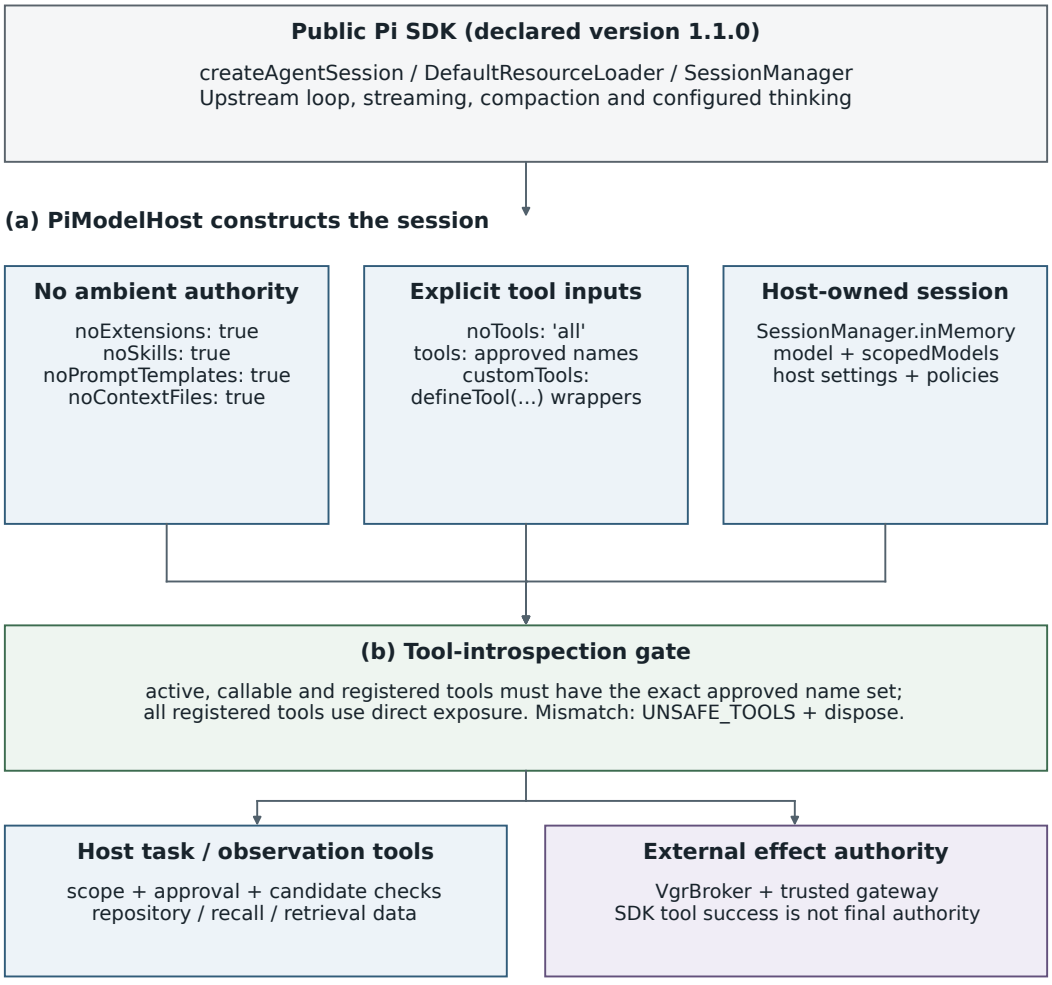


Figure 13: Explicit session capabilities and external effect authority. Pi supplies the reusable session substrate, while the host fixes the callable tool set and rejects mismatches. The native adapter remains outside model-issued permission. This implements a concrete interface boundary for studying control as proposer capability grows.

Figure 13 shows the check that closes a common integration gap: declaring an allowlist is insufficient if the underlying SDK still exposes an ambient tool. Voder compares the active, callable, and

registered name sets against the approved set, rejects duplicates or extra names, and requires direct exposure for registered tools. A mismatch disposes of the session and raises `UNSAFE_TOOLS`. Thus the wrapper checks the assembled session, rather than assuming that its construction arguments were honored.

This capability boundary implements complete mediation at the exposed session interface: proposal computation reaches effectful functionality through host-selected adapters, rather than through ambient tool registration. Its scope is that interface and its trusted construction; broader deployment capabilities require their own mediation. The important control property is explicit effect ownership, not the choice of model-session library [9].

This restriction does not remove Voder's own skills or repository inspection. Such capabilities enter through explicit host interfaces described below. Repository reads and observations provide information; the `VgrBroker` owns the privileged effect path. A successful custom-tool return cannot manufacture a native commit, and an assistant's final message cannot replace commit confirmation. The engineering benefit of reusing Pi is therefore preserved without inheriting an unrestricted default filesystem tool surface.

4.4 Provider configuration and request continuity

Closing the tool surface still leaves the model request itself as a configuration boundary. The model-plugin runtime selects an installed API or weights provider from host configuration, validates explicit model profiles, and constructs an isolated registry without ambient model-file discovery or automatic catalog refresh. Disabling catalog networking here does not disable inference traffic. Endpoints, credentials, context limits, and compatibility options originate in the host configuration; they are not inferred from model-generated text.

Reasoning behavior is also configuration-specific. The primary model declaration in this snapshot sets `reasoning: false` and `thinkingLevel: 'off'`, while additional catalog entries can declare their own reasoning and compatibility policy. The request wrapper forwards enabled reasoning settings where configured. These adapter settings establish what the client requests, not whether a provider performs undisclosed internal reasoning; zero-valued cost metadata likewise does not establish zero inference cost.

For each request, including compaction calls, the guarded stream rechecks the selected provider and model, the exposed tools, and the canonical system rules. Capacity accounting uses a matching successful assistant-usage record when available and estimates the unaccounted tail and tool schemas; it falls back to a character/UTF-8 estimate rather than claiming exact tokenizer counts. Safety, finalization, and output reserves reduce the available context budget. Image payloads are retained and budgeted separately, so base64 transport length is not mistaken for prose-token usage. Requests that exceed a mandatory bound fail explicitly instead of silently dropping protected rules.

Output truncation has a different recovery path from execution failure. The recovery wrapper acts on the observed length stop reason, checks whether continuation is still permitted, and detects repeated partial content or lack of progress. It steers the same task toward completing the interrupted response; incomplete tool arguments must be reconstructed before use. This continuation does not reset native action identity or permit an already committed action to run again. Separating request recovery from effect recovery prevents an ordinary language-model transport problem from becoming a duplicate persistent operation.

Input economy also applies to visual history. The Pi adapter preserves the transcript but omits superseded host-browser images from later provider requests, retaining their text and source identity. It keeps the latest inspection for each path and complementary probes of the same source bytes. User attachments and other paths remain intact. This reduces repeated image payloads without discarding the current evidence needed for review; provider image-token charges remain model-dependent.

4.5 Task admission, scope preparation, and exact edits

With a bounded session available, the editor must establish what that session is allowed to change. The ordinary task entry validates the selected project and its ownership conditions, constructs a plan with exact target files and checks, and requests approval tied to the session, request identity, and scope digest. The host then prepares the task checkout, validator, and baseline. This preparation may create authorized placeholders for new files; it is a host-controlled setup operation, distinct from admitting a model-proposed candidate.

That distinction reconciles new-file tasks with the native gateway's stricter existing-file contract. The model edits the prepared baseline rather than inventing an absent source file inside the native transaction. `loadScope` instantiates the candidate verifier, native transport, commit verifier, and delivery handler for the approved scope, persists the active-scope record, and applies session-reservation checks. An old UI view or historical scope cannot silently become the owner of a new native attempt.

Within that scope, compact edits are expanded against the full pinned preimage. A partial read cannot establish an exact edit base, and an absent source produces a recoverable proposal error directing the session to inspect the actual checkout. Literal patch handling can repair arithmetically incorrect hunk counts without synthesizing context or altering edit payloads. The expanded final bytes, rather than the model's abbreviated representation, become the candidate bound to verification and subsequent dispatch.

This representation also avoids repeatedly emitting an unchanged whole file when a unique, hash-pinned edit can express the change. Expansion is deterministic and read-only; ambiguous matches or a stale base are rejected. For a delivered `.patch` or `.diff` artifact, host Git additionally checks syntax and applicability in the independent checkout before the Modal request is constructed. A rejected preflight returns repair diagnostics with `sandboxed: false`, so malformed patches incur no candidate-check Modal execution and cannot acquire PASS.

The candidate verifier checks those bytes in an independent checkout before the ordinary path permits native submission. This ordering is the relevant engineering distinction from the reviewed DSH default write path: a starting-state compare-and-swap guard protects against a stale write base, whereas candidate isolation prevents an unqualified proposal from becoming the target workspace's accepted replacement. The cited DSH mutation and post-execution pipeline establish the narrower ordering difference; they do not prove that every DSH configuration lacks compensating policy or transaction mechanisms.

In the ASI-control framing, this is a specific engineering intervention against premature effect: the proposal is evaluated against a controlled baseline before the destination adopts it. Approval, isolation, and qualification each retain their own meaning. A convincing plan cannot expand scope, and an isolated execution result cannot bypass native commitment or final delivery checks.

4.6 Native dispatch, sandbox execution, and observation

Exact candidate PASS is a prerequisite for crossing the next boundary, not the result of crossing it. Before spawning the gateway, the host persists a dispatch intent binding the session, action, run, and idempotency identity. Failure to persist that checkpoint prevents dispatch. The transport validates the action schema and approved scope, consumes its attempt identity before process creation, and invokes the pinned gateway without a shell. Process output remains a proposed receipt until its identity and native state have been checked.

Figure 14 distinguishes the SDK checkpoint from the native journal. The gateway owns its worktree and native state directory; the sandbox adapter writes a structured request and invokes the configured Node/Modal driver. Captured output and receipts are checked against the corresponding request and artifact identities. Native evidence admission, obligation closure, and the final meta-audit remain

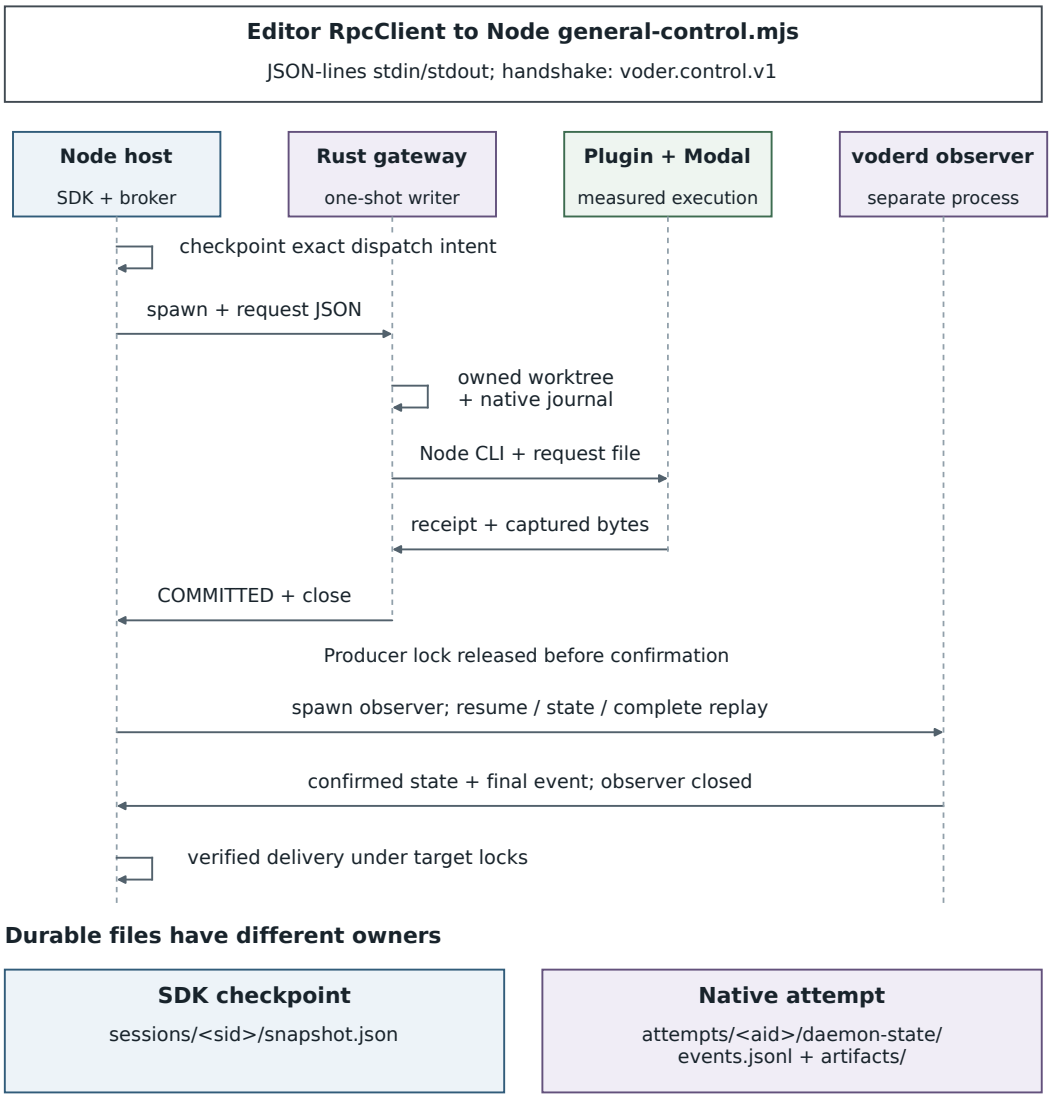


Figure 14: Claim production separated from authoritative confirmation. Exact-candidate PASS precedes this trace; durable intent precedes native dispatch; the producer closes before a separate observer reconstructs state. Only confirmed native authority reaches separately checked delivery. SDK checkpoints and native journals have different owners and meanings.

in Rust, so the host's earlier PASS or the model's explanation cannot substitute for native evidence. The current gateway closes the pinned validator-exit-zero obligation; it does not compile arbitrary natural-language requirements into a complete verifier.

After the gateway process closes, its producer lock must be released before a distinct `voderd` observer opens the journal. The observer resumes the specified session, reads run state, and replays events in bounded pages. Confirmation requires the expected committed state and a unique final-commit event with exact session, run, action, and digest binding. The observer must itself close successfully before confirmation returns. Its read-only role is enforced by the client adapter's exposed protocol methods; a separate process is not a claim of hardware isolation or protection from a compromised host.

This process arrangement carries the logical distinction between a claim and authority into the execution path. The model cannot satisfy it by fabricating a completion message, and the gateway's own response is insufficient without independent observation. Stronger assurance requires preserving the integrity of both producers and observers; the current design makes those dependencies explicit and inspectable.

The sandbox adapter reduces transfer overhead before this evidence is produced. It selects compressed archive transport for at least 32 input files or more than 8 MiB of source bytes. The source manifest is streamed into a hash-bound archive and uploaded in chunks of at most 8 MiB, replacing one payload upload per file with one per chunk. Archive verification, extraction, and per-file checksum checks still run in each independently created sandbox.

For n input files and compressed archive length z bytes, the archive branch uses $\lceil z/(8 \cdot 2^{20}) \rceil$ binary payload uploads, plus checksum metadata and fixed remote verification/extraction operations. This is an RPC-count reduction when the chunk count is smaller than n, not a guaranteed compression ratio or invoice reduction. A bounded 32 MiB process-local cache can avoid reconstructing an unchanged archive, but re-reads and validates source bytes on a hit. Separate CLI processes need not share that cache, and no validator verdict is reused.

Long execution and uncertain execution are treated separately. The transport's legacy foreground-wait parameter is not a universal deadline that automatically kills a valid long task. Explicit cancellation can terminate execution, but termination does not prove rollback. Sandbox calls and recovery observers still have their own configured limits. Once dispatch may have occurred, a timeout, malformed receipt, or broken pipe leaves the attempt fenced until its native outcome is established; it does not authorize a fresh attempt merely because the UI stopped waiting.

4.7 Durable sessions and outcome reconciliation

Reconciling an uncertain outcome also requires a durable host record. The control plane records session snapshots under a private journal root, using owner-restricted directories and files and a checksum for corruption detection. It stores bounded control events and task metadata rather than treating every streamed token as durable state. Event and record limits fail explicitly; the implementation does not silently truncate a record and then present the remainder as a complete session.

The JSON-lines control interface negotiates `voder.control.v1` and exposes session creation, submission, events, cancellation, compaction, model switching, and other capabilities according to the constructed adapter. A historical session view is not a live execution owner. In particular, the legacy resume operation is read-only; recovery requires the explicit native-continuation interface. The configured CLI constructs `NativeSdkContinuation`, whereas the ordinary editor constructor does not thereby acquire that optional recovery capability. Documenting the constructor boundary avoids promising automatic continuation in every frontend.

When installed, native continuation first observes the old action. A committed result is independently confirmed and cannot be rolled back to make retry convenient. A recovered rollback requires its own

unique reconciliation event, a state that disallows delivery, and absence of a final commit. Only independently established rollback permits fresh planning; an unknown state remains blocked. If a commit has been observed but identity, replay, or shutdown checks fail, the adapter reports uncertainty and refuses automatic recovery. This makes recovery a state-reconciliation operation rather than a replay of the last conversational instruction.

Reconciliation avoids paying to regenerate or re-execute a task whose persistent outcome already exists. The kernel can return the confirmed native receipt for the same committed candidate; it refuses a different replacement after commitment. The continuation adapter consults durable native state instead of asking a model to reconstruct execution history. These paths preserve completed work without treating a timeout as evidence that nothing happened; uncertain outcomes still require resolution, not an economical guess.

The same ordering governs checkpoint failures after successful native confirmation. A host unable to record the transition retains the native outcome and withholds delivery; it cannot erase the commit by resetting its local snapshot. Checksums and private files protect this bookkeeping against ordinary corruption and unintended access, but are not cryptographic attestation against the host that owns them. The trusted execution assumptions remain those of Section 3.13.

These retained records also prevent conversational recovery from silently renewing the budget or native-attempt identity. Their checks remain mandatory even when repeating the task would appear faster or cheaper.

4.8 Verified delivery and original-project publication

Once commitment is known, delivery must still establish which files received the verified bytes. The ordinary single-file and batch delivery handlers bind the policy and candidate root, lock the targets, recheck source preimages and candidate bytes, perform replacement, and validate the resulting hashes before recording completion. Batch restoration is restricted to recognized states; an unexplained external modification cannot be overwritten under the pretext of restoring an earlier candidate.

Figure 15 addresses the harder case in which the user selects a privately writable directory inside a larger Git repository. Voder creates an owned working copy, excludes the original Git metadata, initializes a private baseline, and records source hashes in a sidecar. Native execution and verified delivery operate on that copy. Ordinary task entry rejects a group- or other-writable selected root before this helper is reached; private-copy support must not be interpreted as a bypass of that entry check. Read-only tasks do not require the copy path.

Publishing the copy back to the selected project is a separate approved operation. The approval identifies destination files and final hashes. After approval, the publisher rechecks both the private output bytes and the original baseline, creates a fresh randomized transaction directory, and persists an intent record. Existing-file replacement uses rename, while new files use no-clobber linking; directory synchronization and post-write hash checks precede the delivered record. Fresh transaction directories distinguish concurrent or repeated publication attempts without relying on a shared session-named directory being absent.

This protocol does not make a multi-file publication globally atomic. A failure can leave a partial original-project update, in which case the failed record and verified private output are retained and the original project is not reported as delivered. The system must not rerun a committed native action to repair a publication-only failure. Native commitment, verified working-copy delivery, original-project publication, and a Git commit are therefore distinct events, even when the interface presents them as stages of one task.

Publication consequently has its own destination authority rather than inheriting unlimited permission from an earlier successful run. This closes the identity chain at the selected project and prevents a private success from being misreported as an external delivery. Future non-filesystem integrations

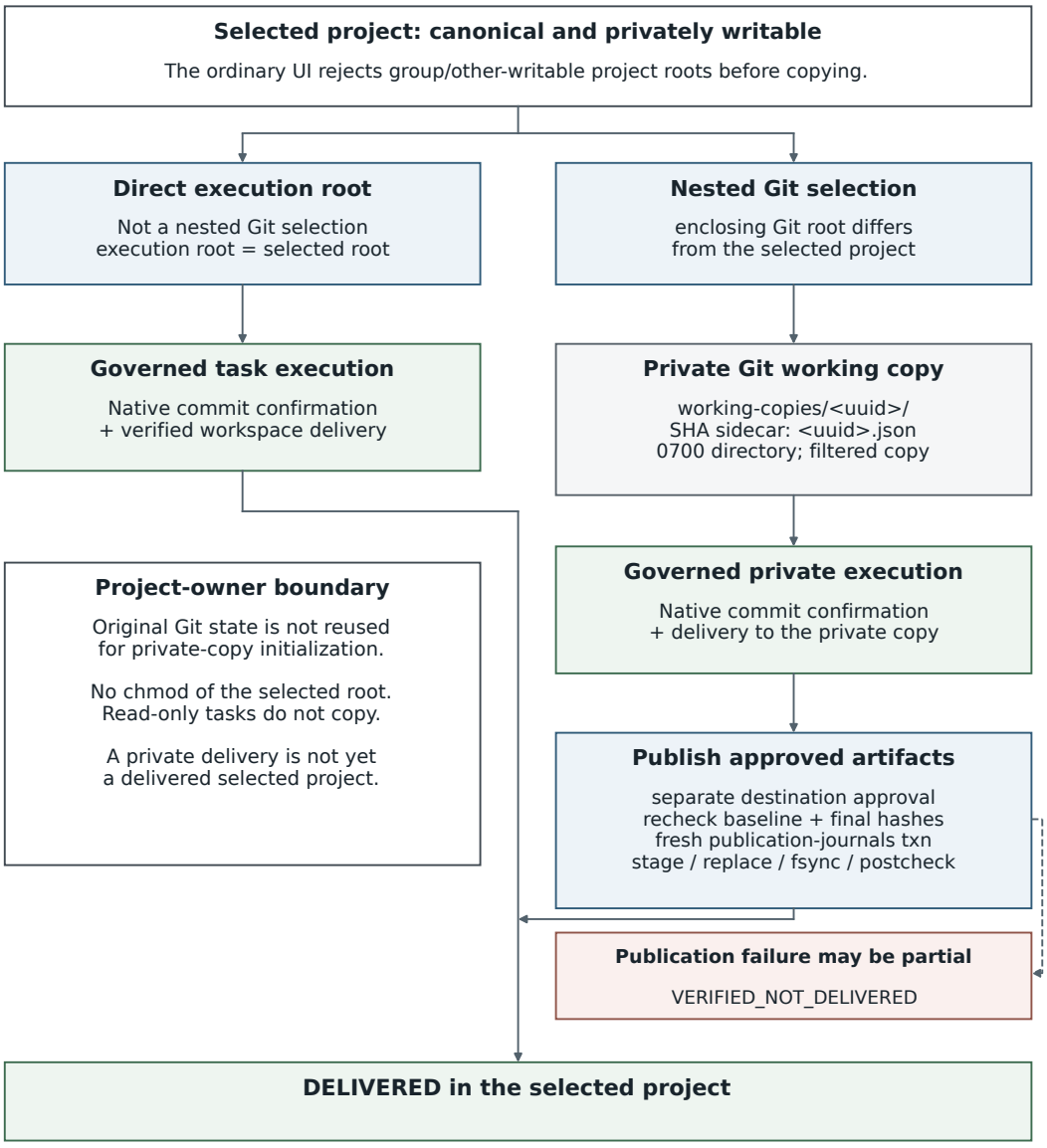


Figure 15: Destination-specific authority beyond private execution. A nested selection uses a private working copy and source baseline. Verified private delivery precedes separate publication approval and current destination checks. A partial publication failure retains evidence without claiming delivery or replaying native work.

must establish the analogous effect-specific boundary instead of assuming that a code rollback can reverse their consequences.

Separating publication also contains recovery cost. A destination conflict can be investigated against the retained private output and intent record without rerunning generation and native validation solely to recreate those bytes. Any eventual publication must still satisfy its approval, baseline, and final-hash conditions. Retaining a verified artifact saves recomputation; it does not authorize overwriting an independently changed destination.

4.9 Context policy, skills, and engineering memory

Keeping those stages distinct also requires controlling what the model carries between them. The default context provider loads host-bundled policies and selected skills from manifests with pinned content hashes. It checks regular-file identity, location, encoding, and size before injection. Canonical rules remain in the main system prompt, while stage-specific overlays serve next-action, repair, preflight, and delivery processing. The preflight and delivery preparation interfaces inspect policy identity without issuing an additional model call or granting native authority.

Skill selection combines explicit configuration with bounded task-term and host-signal matching. Automatically selected bodies may be omitted when the supplement budget is exhausted; an explicitly required body or mandatory policy that cannot fit causes an error. No body is silently clipped and presented as complete. This implementation is distinct from both Pi's disabled ambient skill loader and the optional `InstalledSkills` store. The ordinary SDK uses the bundled provider, while the editor constructor does not pass the separate optional store. Treating these as three different loading paths explains how Voder can reuse Pi while replacing its implicit resource discovery.

The economy comes from deciding relevance locally. Policy preparation and lexical skill selection do not require a separate planning model invocation, and only selected bodies consume the bounded supplement. Mandatory rules remain present, while an explicitly required skill that cannot fit causes an error. This avoids the false economy of truncating a required instruction merely to reduce input size.

Project memory follows the same distinction between retained data and active context. Conversation pairs are archived with source locations, indexed for retrieval, and selected by the project-memory policy rather than reinjected wholesale at startup. The policy supports automatic, disabled, and selected-only modes, exact phrase controls, term matching, and recency, with bounded whole-block injection. Recent-history, search, and source-reading operations serve different retrieval needs; source-line checks bind an excerpt to the retained dialogue instead of treating an index hit as the complete record.

Portable memory is a separate user-data mechanism. The model may propose entries, but export requires explicit selected identities and user confirmation. The bundle validates schema, field bounds, and a canonical checksum, with a defensive check for recognizable secrets. That checksum detects changes relative to the exported bytes; it is not a signature, and the secret check is not a complete data-loss-prevention system. Imported memory remains contextual material and cannot supply a permit, validator identity, or native evidence record.

Archival storage and inference expenditure can consequently be managed independently. Recent recall restores the relevant completed dialogue pairs after a refresh, while source-based paging retrieves detail only when needed. The archive is not paid for again in full merely because it exists. A smaller selected payload is a concrete reduction relative to whole-history injection, although actual input-token and monetary savings require the chosen provider's usage and billing records. Section 5.5 measures the selected payload on controlled fixtures.

4.10 Read-only tools, retrieval, and planning aids

Selective context becomes useful only if the session can obtain information not yet available through equally explicit interfaces. `HostToolRegistry` supplies host-approved read capabilities, while `ToolEconomy` tracks bounded invocation behavior, repeated failures, and redundant work. Tool declarations and input schemas are checked before exposure. These mechanisms regulate observation cost and availability; they do not extend the native action allowlist or turn retrieved text into execution authority.

The search service separates local knowledge and repository lookup from configured web-provider routes. Retrieved passages carry source and content identity so that a claim can be traced to material actually returned by a provider. A discovery error, timeout, or empty result remains an information failure rather than evidence for an answer. Provider adapters make retrieval replaceable, but a source hash proves byte identity, not the truth of the page, and a search result is not the authenticated validator evidence admitted by Rust.

The task planner is similarly an advisor rather than another autonomous agent loop. Its deterministic scorer ranks inspection, diagnosis, patching, verification, or stopping using fixed information, cost, risk, and stage-fit terms. The task plan tracks observations, failures, revisions, and consumed native actions, and emits a recommendation and compact stage description. These are implementation heuristics, not learned utilities or a proof of globally optimal scheduling.

The ordinary host also submits the original task without a mandatory preliminary model response containing a planning JSON document. Local preparation and deterministic planning can therefore avoid that auxiliary generation round; the agent may still reason and inspect as the task requires.

The Rust utility-ranking and semantic-retry libraries discussed in Section 3 remain separate from this host policy; they are not integrated into ordinary native authorization.

4.11 Reference inputs and final-source visual inspection

The same observation boundary extends to user-provided files. Reference blocks distinguish workspace files, editor selections, and attachments, and validate their identity, digest, type, and optional source range. Image inputs require canonical encoding and recognized content signatures. The reference contract is additive: an attachment cannot declare required target files, executable actions, or authority simply by containing fields or instructions that resemble host configuration.

Document conversion runs an explicit decoder process without a shell, with bounded output and diagnostics and a cancellation path. Office material can be converted to PDF, while PDF and image handling preserve a visual route alongside extracted text. Invalid conversion output or unavailable dependencies produce an explicit error instead of sending an incomplete document as if parsing had succeeded. Archive handling likewise distinguishes inline parsing from extraction, so a resource-bound failure does not silently pass only the convenient prefix of an archive to the model.

Input inspection and output acceptance remain separate. For generated HTML or SVG, the browser and renderer helpers inspect an artifact associated with the final candidate source; the final-source visual gate described in Section 3 checks that binding before delivery. A screenshot from an earlier draft cannot qualify a later file solely because the session remembers seeing a similar page. Render completion still establishes neither subjective design quality nor full functional correctness. Its engineering role is to make a required visual observation refer to the bytes being delivered.

Browser inspection has a more direct paid-execution saving. The ordinary host keys retained frames by the normalized full request and source digest and requires the same current evidence-row identity. Before an ACCEPT review, an eligible repeated request can reattach the real captured images at most twice, returning `frames_reused: true` before a new browser is launched. Subsequent covered repetitions require review rather than another identical paid inspection.

Each such replay performs zero new remote browser executions. It may still consume model input and image tokens, so replay is not described as free inference. A changed request, different source, replaced evidence row, or already accepted review does not qualify for this replay branch; normal inspection and readiness gates then apply. No replay produces visual ACCEPT automatically, and complementary novel probes remain available when a concrete requirement lacks evidence.

4.12 Operator interfaces, diagnostics, and release scope

The inspected VS Code extension identifies itself as version `0.1.45` in its source package and release VSIX manifest; this is a packaging version, not a native-authorization version. These internal distinctions must survive presentation to the operator. The editor resolves task scope, displays approvals, streams progress, and presents candidate or delivery results, while the control process owns execution state. The CLI exposes explicit native configuration and a draft mode; an unconfigured run must be requested as an unverified draft rather than acquiring native authority from a plausible-looking JSON response. Cancellation, disconnection, historical replay, and successful delivery are separate control events, not interchangeable indications that the task has finished.

For direct source use, an operator first configures the provider and native policy, then supplies a private gateway-configuration file naming `binary`, `daemonBinary`, `policyFile`, `attemptStore`, `nodeBinary`, `modalCli`, and `targetPath`, with optional `targetPaths`. The CLI rejects a symlink, non-regular file, excessive configuration size, unknown keys, or group/other access. From the checkout root, the following invocation illustrates the verified entry after those prerequisites have been satisfied; the absolute paths and goal are deployment-specific placeholders, not a preconfigured runnable task.

```
node voder-agent/packages/voder-cli/bin/voder-cli.mjs run \
  --workspace /absolute/project \
  --gateway-config /absolute/private/gateway.json \
  --goal 'Apply the approved scoped change' --format json
```

The same module's `rpc` command serves JSON-RPC over standard-input/output lines; `status`, `events`, and `resume` inspect retained state. An explicit `--draft` request produces unverified text and cannot be combined with native configuration. Consumers should branch on the returned structured state and its destination semantics, rather than infer success from the assistant's wording.

Diagnostics retain classified failures and bounded observations for repair and review. The acceptance package contains fixtures, audit, live-execution, and release-check modules, which provide distinct validation surfaces. Their presence in the archive is not evidence that every test has passed on every deployment or that a benchmark was reproduced during preparation of this report. Likewise, `ImprovementCandidates` can retain quarantined method proposals and qualify independently evaluated candidates for human review, but does not automatically activate them in the ordinary authorization path.

A source-to-role traceability index appears in Appendix A. The implementation's principal engineering result is therefore a traceable route from approved scope to exact candidate checking, durable native dispatch, independent outcome confirmation, and destination-specific delivery. Provider adapters, Pi session reuse, contextual skills, memory, and visual tooling make that route usable without becoming alternate sources of permission.

Accounting for avoided work. These mechanisms expose operation-level savings without conflating them with a measured end-to-end discount. For the accounting comparison, let N_{pf} count candidate-check sandbox executions for a locally rejected patch, N_{br} count new browser executions for an eligible frame replay, and N_{dup} count verifier callbacks for an already-probed candidate. The

respective implemented branches satisfy

$$N_{\text{pf}} = 0, \qquad N_{\text{br}} = 0, \qquad N_{\text{dup}} = 0. \tag{4.1}$$

Each count concerns its named rejected or replayed request, not the initial inspection, every tool call, or the native validator. The previously required checks and delivery predicates remain unchanged. Relative to executing the corresponding remote operation again, these are eliminated executions rather than a lower-quality substitute.

To separate that structural result from billing, let $\mathcal{O}$ be operations avoided by these branches, c_o the counterfactual charge for operation $o \in \mathcal{O}$, and c_{local} and c_{follow} the incremental charges for local guards and subsequent recovery interactions. A like-for-like accounting decomposition is

$$\Delta C = \sum_{o \in \mathcal{O}} c_o - c_{\text{local}} - c_{\text{follow}}. \tag{4.2}$$

Here ΔC is the counterfactual baseline charge minus the revised-path charge, in one currency; each avoided invocation is counted once. Shared work cancels in this comparison. Source inspection establishes the skipped operations, not these charges or positive net savings for every workload. Neither equation is an online controller or a relaxation of VGR. A deployment-level savings percentage requires matched task outcomes, provider usage, sandbox billing, and recovery cost; none is inferred from zero-valued model cost metadata.

Current limits remain explicit: deployment components are resolved separately, the gateway closes one configured success obligation, optional recovery depends on its constructor, original-project publication can fail partially, and experimental decision or improvement libraries are not claimed as production integrations. These boundaries delimit what the public source is intended to make inspectable and reproducible, rather than replacing implementation evidence with a broader claim of autonomy.

5 Empirical Evaluation

Appendix A maps the main authority boundaries to inspected source components, while Appendix B documents the supplied experimental evidence and score policy. The engineering boundaries established above lead to three distinct evaluation questions: whether the implementation preserves its local invariants, how much context and orchestration work it requires, and which benchmark outcomes are supported by retained evidence. We examine these questions separately so that a component assertion, a synthetic efficiency measurement, and a task reward cannot be mistaken for the same kind of success.

5.1 Evaluation protocol and evidence provenance

The evaluation combines two author-supplied evidence collections: offline engineering experiments and a separate four-suite benchmark reward audit. We recomputed the tabulations from the supplied logs, manifests, and per-task audit records; no model generation, sandbox execution, or benchmark task was rerun during manuscript preparation. Throughout Section 5, the sole author-reported model configuration for any model-connected experiment, including all four benchmark suites, is **DeepSeek V4.1 Flash 1017** through Voder; GPT-6 Astra is not part of this evaluation. Component regressions, concurrency stress, recovery fixtures, and context microbenchmarks instead exercise local Voder implementations without invoking a model API. Their assertion counts, memory estimates, and timings therefore measure engineering properties of the same harness, not DeepSeek inference performance. The model-configuration attribution for benchmark tasks is author-reported rather than independently established from complete per-task provider traces.

The archive checks covered 41 listed file hashes across seven distinct nested archives. Eight source/test-file hashes in the engineering closure record also match the source snapshot used in Sections 3–4.

The baseline Node battery and process-concurrency experiment used Node 22.21.1. The later Rust collection records Cargo 1.99.0; the memory microbenchmarks and engineering-closure tests used Node 24.13.0. These are separate runs, not one hardware-normalized experiment. The records do not provide a sufficiently complete hardware and scheduling description to support cross-machine latency comparisons.

Counts retain their original units. The 89 Node and 229 Rust results count test cases within component suites; the concurrency experiment counts separate process executions. Repeated suites, the seven-process retest, and supplementary regressions are reported separately rather than added to a purported total of independent tests. The offline engineering runs record zero model calls, zero Modal calls, and unchanged production-source hashes.

5.2 Component verification and concurrency stress

The first question is whether the local mechanisms underlying the governed path behave as specified in the tested cases. Table 1 reports the baseline assertion counts recovered from the raw Node logs and Rust run records. All 89 Node cases and all 229 Rust cases passed in those recorded runs. The Rust result comes from the later toolchain-enabled collection; the earlier collection's four unavailable Rust groups remain recorded as blocked, rather than being reclassified as passes.

Table 1: Baseline component test results. Counts refer to recorded test cases, not agent tasks.

Component	Passed / tested
Host scheduling	12 / 12
Host finalization	9 / 9
Candidate/source identity	9 / 9
Verifier budgeting	4 / 4
Native observation adapter	5 / 5
Delivery recovery	10 / 10
Native journal interface	1 / 1
Read-only tool regulation	7 / 7
Native gateway contract	8 / 8
Project memory	16 / 16
Task/Git boundary	4 / 4
Acceptance fixtures	4 / 4
Node subtotal	**89 / 89**
Rust core	170 / 170
Rust runtime library	23 / 23
Rust daemon library	22 / 22
Rust protocol library	14 / 14
Rust subtotal	**229 / 229**

The stress experiment then repeats seven component groups—host scheduling, finalization, candidate pinning, verifier budgeting, native observation, delivery recovery, and journaling—at process concurrency levels 1, 2, and 4. Each group is executed 20 times per level, yielding 140 runs per level and 420 runs overall. All recorded runs passed, with no failed, timed-out, or blocked stress execution.

Table 2 shows batch throughput increasing from 2.321 to 9.137 process runs/s over these configurations, an observed ratio of approximately 3.94. The work consists of isolated component processes, not concurrent model-driven tasks. This result therefore supports repeatability and local process-level scaling under the tested schedule, not a 3.94-fold acceleration of agent task completion. The separate seven-process retest also passed and is not an additional set of unique assertions.

Table 2: Process-concurrency stress results: seven component groups, 20 runs per group at each level.

Concurrency	Passed / runs	Batch time (s)	Runs/s
1	140 / 140	60.318	2.321
2	140 / 140	30.170	4.640
4	140 / 140	15.322	9.137

The separate seven-process retest is excluded. Throughput is component-process throughput, not live-agent throughput.

Control interpretation. The component and stress results support the repeatability of guards that connect candidate identity, scheduling, observation, and recovery. Within these fixtures, increasing process concurrency did not produce a recorded failed execution. This is evidence about implementation behavior under the tested schedule, rather than a population-level failure bound or an adversarial ASI-control result. For increasingly capable agents, the relevant design property is that higher proposal throughput does not confer additional acceptance authority. These measurements provide bounded engineering support for retaining the same external control conditions as useful execution scales; stronger proposers can be evaluated against that boundary without granting them permission to redefine it.

5.3 Native authority and delivery validation

Passing component tests is most informative when their acceptance conditions align with the authority boundary. The focused checks in Table 3 exercise fabricated or mismatched commit claims, observer lock ownership, and verified workspace delivery. Their fixtures deliberately isolate these interfaces; a passing fixture must not be read as an end-to-end native transaction.

Table 3: Focused authority and delivery checks. All rows describe isolated fixtures.

Suite	Passed / tested	Execution boundary
Commit verification	7 / 7	Controlled observer/evidence fixtures
Observer locking	5 / 5	OS lock; Python protocol stand-in
Verified delivery	10 / 10	Temporary filesystem workspaces
SDK crash recovery (final log)	11 / 11	Child SIGKILL; native stand-ins
SDK recovery (historical)	10 / 11	Earlier assertion; failure retained

These suites are not summed with the baseline or supplementary runs. The final SDK log supersedes only the revised assertion, not the historical record.

The commit-verifier cases reject fabricated committed state, wrong action identity, absent or duplicate final events, and transport failures. The observer-lock cases use an actual OS lock with a Python protocol stand-in, rather than the Rust daemon. Delivery checks use temporary workspaces to exercise candidate identity, stale-source conflicts, and restoration boundaries. These results test the guards that prevent an invalid success claim, while leaving live-model behavior and remote execution outside their scope.

Control interpretation. These negative cases most directly test the authority gap in Figure 2: fabricated success, mismatched identity, or incomplete observation must not qualify as confirmed commitment. Delivery conflicts test the corresponding destination boundary. Their reported rejection shows that, in these fixtures, a producer cannot turn a success claim into accepted state by fabricating a verdict or substituting an action identity. This is a concrete prerequisite for studying more capable proposers under externally retained authority: acceptance follows independently checked identity and evidence rather than the persuasiveness of generated content. Coverage of harmful program semantics and trusted-host compromise remains a separate assurance obligation.

The supplementary microbenchmark runner also records 15 context-capacity and six context-lifecycle regressions, together with another seven-case commit-verifier run. These 28 results are retained as

supplementary runs. They overlap existing mechanisms and repeated suites, so we do not add them to the baseline assertion totals.

5.4 Crash recovery and durable-state replay

The next boundary is interruption after an execution identity has been persisted. The SDK recovery battery contains an actual child-process SIGKILL fixture with controlled native-continuation stand-ins. The final attached log reports 11/11 passing cases, including refusal to replay a committed or uncertain action, cancellation persistence, task-identity checks, competing recovery coordinators, and bounded automatic replanning.

The evidence also retains an earlier 10/11 SDK run. The later record supersedes the earlier result for the revised test assertion; it is not evidence that a production-source repair was made. Both records remain part of the reported history. The final log alone does not establish a fresh source-hash attestation for the revised test file or validate an in-flight Rust commit under failure.

Table 4: Real Rust daemon restart after durable acknowledgment. All three cases preserve session identity and idempotent replay.

Case	State before kill	Events before / after	Result	Case time (s)
1	BLOCKED	2 / 2	PASS	0.0438
2	BLOCKED	2 / 2	PASS	0.0364
3	BLOCKED	2 / 2	PASS	0.0408

The acknowledged state is BLOCKED, not an in-flight VGR commit. Case time includes the full test sequence.

A separate experiment uses the real `voderd` executable. In all three cases, the task had already reached a durably acknowledged `BLOCKED` state with two journal events before SIGKILL. After restart, the session and blocked task persisted, replay returned the same events, and no duplicate event was appended. Table 4 reports the individual case durations as recorded; these are whole-case elapsed times, not isolated recovery-latency estimates.

This distinction limits the recovery claim precisely. The daemon experiment validates post-acknowledgment persistence and idempotent replay of the tested blocked state. It does not interrupt a VGR write between dispatch and final commitment, and consequently does not demonstrate in-flight native commit crash recovery.

Control interpretation. The revised SDK cases and acknowledged-state daemon restarts support a conservative recovery principle: interrupted communication does not itself renew execution authority, and durable state remains an input to reconciliation. The 11/11 SDK and 3/3 daemon results cover different fixtures and cannot be pooled into a single end-to-end reliability estimate. Together they support continuity of the acceptance boundary across the tested disruptions: loss of communication is not converted into renewed permission, and acknowledged blocked state remains blocked after restart. Preserving such restrictions over long trajectories is directly relevant to controlled agent development. In-flight fault injection is the next evidence needed to extend this argument across commitment phases.

5.5 Context selection and estimated token reduction

With the correctness scope fixed, we turn to the contextual economy described in Sections 3–4. The first microbenchmark calls the production `ProjectConversationMemory.compose` API on deterministic synthetic histories of 100, 500, and 1,000 records using the query `gateway validation`. Its counterfactual baseline concatenates every record in the same entry format. Full-history injection is an experimental comparator, not a claimed Voder production configuration.

Table 5: Memory selection against a constructed full-history baseline. Query: gateway validation; three synthetic fixture sizes.

Rows	Full-history bytes	Selected bytes	Full-history token est.	Selected token est.	Est. reduction (%)
100	46,614	3,122	13,319	893	93.3
500	233,518	6,051	66,720	1,730	97.4
1,000	467,127	6,051	133,465	1,730	98.7

Token values are source-heuristic estimates; byte sizes are exact UTF-8 payload lengths. This is not provider usage or a billing experiment.

Table 5 distinguishes exact UTF-8 payload sizes from source-heuristic token estimates. The selected payload grows from 3,122 bytes at 100 rows to 6,051 bytes at 500 rows, and remains 6,051 bytes at 1,000 rows. The corresponding estimated reduction relative to the constructed full-history baseline is 93.3%, 97.4%, and 98.7%. This measures how much historical material is selected for inference, not task-quality retention or billed-token savings.

Composition latency is measured locally over 80 iterations per size. Table 8 reports its median and P95 alongside the other isolated API measurements. Larger archives therefore need not produce proportionally larger injected context, although selection still has a local processing cost. No model request is issued in this experiment. In the architecture, this bounded selection supplies historical observations without issuing new write authority. The result therefore supports the feasibility of long-running contextual continuity alongside a separate acceptance boundary; it does not measure semantic memory reliability or prove safety from reduced context volume.

5.6 Context-policy ablation

The closure experiment tests whether the reduction depends on the actual selection policy rather than on a substituted memory implementation. It crosses three history sizes with three query classes and three validated policy settings, producing 27 scenarios. The default policy uses `limit=12` and `maxBytes=24576`; the tighter policy uses `limit=6` and `maxBytes=8192`. An off policy is included solely as a negative control.

Table 6: Policy ablation: estimated context volume for all nonempty size/query settings.

Rows	Query	Full-history estimate	Default selected	Tighter selected	Default reduction (%)	Tighter reduction (%)
100	G	14,327	962	962	93.3	93.3
100	M	14,565	847	847	94.2	94.2
100	F	14,587	870	870	94.0	94.0
500	G	71,763	1,867	964	97.4	98.7
500	M	72,951	1,641	849	97.8	98.8
500	F	73,063	1,687	872	97.7	98.8
1,000	G	143,550	1,867	964	98.7	99.3
1,000	M	145,949	1,641	849	98.9	99.4
1,000	F	146,168	1,687	872	98.8	99.4

G: gateway validation; M: memory policy; F: fault recovery. Default: limit 12, budget 24,576 bytes. Tighter: limit 6, budget 8,192 bytes. All nine off-policy controls return zero content and are excluded as optimizations.

Table 6 reports all nine size/query pairs for the two nonempty policies. At 100 rows, the default and tighter policies select the same material in each pair. At 500 and 1,000 rows, the tighter policy selects fewer estimated tokens. For the 1,000-row fixtures, reductions relative to full-history injection range from 98.7% to 98.9% under the default policy and from 99.3% to 99.4% under the tighter policy.

These results identify a policy-controlled context-volume tradeoff, not a demonstrated improvement in task utility. At 1,000 rows, tightening the policy reduces the selected estimate by approximately 48.3–48.4% relative to the default setting, depending on query. Whether the omitted material is

necessary for a real task is not measured. The off policy returns zero content in all nine controls and is not treated as a usable 100% saving.

Table 7: Local composition latency for the same policy-ablation pairs (milliseconds).

Rows	Query	Default P50	Default P95	Tighter P50	Tighter P95
100	G	0.7141	1.0987	0.7015	1.1876
100	M	0.6740	1.0082	0.6635	0.9211
100	F	0.6446	0.8111	0.6495	0.8367
500	G	3.2207	3.3492	3.1866	3.2707
500	M	3.2573	4.3697	3.0675	3.2551
500	F	3.0875	3.2877	3.0550	3.2596
1,000	G	6.0331	6.3426	3.5274	4.7161
1,000	M	3.4802	3.7744	3.4578	4.0237
1,000	F	3.4549	3.6490	3.4377	3.6401

Each setting uses 60 measured iterations after eight warmups. G/M/F query definitions match Table 6. Results are descriptive measurements from one host.

Table 7 preserves the paired local latency measurements: 60 iterations after eight warmups per setting. The default gateway-validation case at 1,000 rows has a 6.0331 ms median and 6.3426 ms P95, while the corresponding tighter setting records 3.5274 ms and 4.7161 ms. Other pairs show smaller or mixed changes. These measurements are descriptive; the supplied results do not support a significance test or a general claim that a tighter budget always runs faster.

5.7 Isolated API latency

The remaining microbenchmarks separate bookkeeping from external execution. Table 8 reports the source APIs' measured local overhead, retaining each experiment's iteration count and fixture type. The capacity-accounting fixture supplies a usage anchor of 301,113 tokens and obtains a computed context value of 301,125 after the tail estimate. That anchor is injected test data, not telemetry from a live provider call.

Table 8: Isolated source-API timing measurements (milliseconds).

API / fixture	Iterations	P50	P95
Memory compose (100 rows)	80	0.6515	1.1256
Memory compose (500 rows)	80	2.7912	3.3746
Memory compose (1,000 rows)	80	3.2446	4.4488
Context-capacity accounting	60	0.0017	0.0021
Memory-policy isolation	500	0.0013	0.0016
Commit confirmation (in-memory)	100	0.0038	0.0189

The first three rows are the baseline context microbenchmark, not the later ablation corpus. Commit confirmation excludes Rust daemon startup/I/O, remote execution, and actual recovery.

The policy-isolation fixture exercises a positive and a negative selection case. The commit-verifier fixture accepts matching evidence and rejects an identity mismatch using an in-memory observer. Its microsecond-scale measurements exclude daemon startup, journal I/O, Modal execution, and real recovery. They cannot be substituted for native commit latency. Likewise, the measured context-volume reductions do not establish an invoice-level saving without provider usage, matched task outcomes, and recovery costs.

5.8 Audited benchmark outcomes

Task outcomes are evaluated separately from the offline engineering experiments. The supplied reward audit covers DeepSWE v1.1, Terminal-Bench 3.0, Terminal-Bench 4.0, and SWE-Marathon.

All four are author-reported as Voder runs using the **DeepSeek V4.1 Flash 1017 API**; no GPT-6 Astra benchmark result is included. The audit independently checks retained task rewards, but does not establish a complete per-task model-configuration or generation trace.

Each score uses the fixed full-suite denominator. Only a retained numeric `official_reward`=1 contributes one confirmed pass per task; repeated records do not add successes. The full task population, rather than the number of retained result files, determines the denominator. The selected task records show no contradictory reward observations.

Table 9: Voder benchmark outcomes from the retained reward audit. Score = pass / full-suite size, rounded to one decimal place.

Suite	Pass	Full suite	Score (%)
DeepSWE v1.1	89	113	78.8
Terminal-Bench 3.0	31	74	41.9
Terminal-Bench 4.0	30	66	45.5
SWE-Marathon	9	20	45.0

SWE-Marathon is a reference-assisted internal control. These local audited scores are not asserted as official leaderboard submissions.

Table 9 reports pass counts, full-suite sizes, and scores rounded to one decimal place. The source audit lists 123, 43, 41, and 18 matching reward-record occurrences respectively; task-level deduplication yields the displayed counts. Scores use the complete suite denominator rather than conditioning on the availability of retained reward records.

The SWE-Marathon run has reference-assisted internal-control provenance. Its 45.0% score is therefore not an unassisted model result and is not directly comparable to a standard unassisted leaderboard entry. More generally, these are local audited counts from retained evaluator records, not a claim of accepted official leaderboard submissions. The package contains reward values, evidence paths, and file hashes, but not the underlying evaluator JSON files; reconstruction of their contents would require those original files. Within that provenance boundary, task rewards document useful capability delivered through the reported system, while the authority and recovery fixtures examine its control mechanisms. Their joint research value is the ability to investigate useful performance and independently enforced acceptance in the same architecture. The benchmark scores themselves do not measure control against stronger or adversarial proposers.

5.9 Validity and attribution boundaries

The evidence supports three bounded findings. The recorded component and concurrency cases pass their specified checks; synthetic memory selection substantially reduces the amount of history presented to inference; and the benchmark audit establishes the retained pass counts under a fixed-denominator policy. None of these findings substitutes for another: local guard tests do not predict task success, and a task reward does not validate every recovery invariant.

For the ASI-agent research objective, these findings occupy different roles: guard and recovery tests examine control conditions, task rewards examine useful problem solving, and memory measurements examine contextual cost. Keeping them separate avoids treating stronger capability as evidence of stronger control. The retained record supports the present governance substrate and identifies targeted assurance work; evaluating an adaptive adversarial proposer and recursive agent modification requires additional experiments [6].

The principal open measurements are live-provider billing, task-quality effects of tighter memory selection, complete per-task generation provenance, and interruption during an actual in-flight native commit. The supplied experiments do not resolve them. Descriptive fixture timings and repeated

component runs also do not justify population-level reliability estimates or confidence intervals that treat repetitions as independent task samples.

These boundaries allow the engineering claims to remain reproducible without overstating system capability. Qualitative 3D artifacts, including rendered scenes and interactive applications, require a different presentation and acceptance protocol and are reserved for the subsequent case-study section rather than included in the quantitative benchmark table.

6 Interactive 3D Application Case Studies

Unlike the single-model, DeepSeek-configured evaluation in Section 5, the visual demonstrations in Section 6 use two model APIs: **DeepSeek V4.1 Flash 1017** and **GPT-6 Astra**. Five application case studies examine Voder's visual and interactive delivery. The cycling pelican is a standalone DeepSeek demonstration; the remaining four tasks compare applications built with each API under author-reported identical prompts. Section 6.6 specifies the rendering protocol and interpretation limits; Appendix C indexes the local runnable artifacts.
For the present control question, these applications illustrate the useful artifacts that a governed execution substrate is intended to support. Their visual quality is assessed separately from the source-bound acceptance conditions in Sections 3.9 and 3.8; an appealing rendering is not evidence that those conditions can be omitted.

6.1 Voder capability demonstration: cycling pelican

The cycling-pelican task examines Voder's ability to deliver an integrated animated 3D application using DeepSeek V4.1 Flash 1017. Figure 16 shows the resulting rider, bicycle, and coastal promenade. The delivered HTML combines procedural geometry, lighting, and coordinated articulation in one runnable application.
In this implementation, a common crank phase determines pedal positions, a two-segment inverse-kinematics construction positions the legs, and a fixed gear ratio couples wheel rotation to the crank. At the captured phase, the diagnostic state reports opposed pedals and reachable leg targets; the rendered rider remains seated with its wings directed toward the handlebar. The result demonstrates Voder's ability to integrate procedural modeling, articulated motion, and scene rendering into one runnable character-animation application using the specified model.



Figure 16: Standalone Voder capability demonstration using DeepSeek V4.1 Flash 1017: a cycling pelican rendered at crank phase 0.7 radians. Interactive HTML.

6.2 Scripted reusable-booster recovery

The booster case extends those local relationships into a timed sequence involving a vehicle, a tower, articulated catching arms, exhaust, and a moving camera. Both implementations define a 30-second presentation with approach, alignment, capture, and a final held pose. We inspect the same timeline samples at 6, 18, and 24 seconds; Figure 17 presents the common post-cutoff state at 24 seconds, when neither implementation's thrust function remains active. This phase offers a comparable view of the delivered recovery composition without conflating ignition brightness with structural detail.

(a) DeepSeek V4.1 Flash 1017 (b) GPT-6 Astra

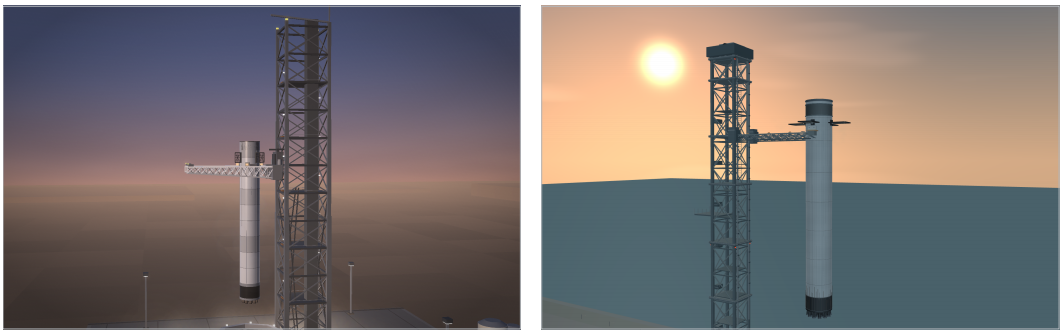


Figure 17: Voder-produced reusable-booster recovery at $t = 24$ seconds in both timelines, after scripted engine cutoff. Native camera paths and vehicle proportions are retained; panel labels identify the model configuration. Interactive HTML.

The DeepSeek-backed output gives greater prominence to metallic highlights, site equipment, and the changing atmospheric background. The Astra-backed output presents a more sparsely furnished site with a clearly separated tower, booster, and articulated grid fins. Both make the intended recovery event recognizable, while differing in staging and scene density. Their catches are prescribed by animation functions: the images do not validate contact dynamics, structural load transfer, or a physically controlled landing. The sampled states support a comparison of cinematic delivery, not a claim of aerospace simulation fidelity.

6.3 Shader-based black-hole visualization

Where the booster coordinates explicit scene objects, the black-hole case concentrates much of its visual construction in a fragment shader. Both outputs combine a dark central silhouette, a bright accretion structure, an upper arc, and a star field, with orbit and brightness controls in the application. For Figure 18, we hold shader time at 6 seconds, brightness at its neutral value, and camera yaw and elevation at 0.35 and 0.56 radians, respectively. Each implementation retains its own camera distance and visual scale.

(a) DeepSeek V4.1 Flash 1017 (b) GPT-6 Astra

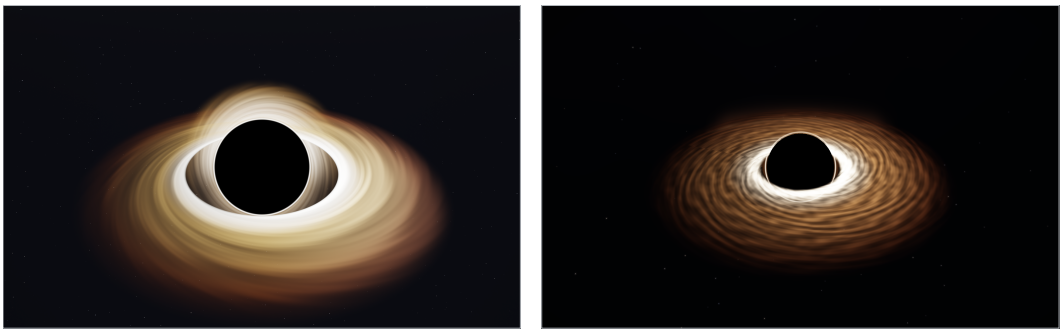


Figure 18: Voder-produced black-hole visualizations at shader time 6 seconds, neutral brightness, and aligned viewing directions. Native camera distances and visual scales are retained. Interactive HTML.

The DeepSeek-backed artifact fills more of the viewport and emphasizes a broad luminous arc around the central silhouette. The Astra-backed artifact exposes more of the surrounding field and separates the outer disk into finer visible bands. These are different presentation choices: the former emphasizes the overall silhouette, while the latter makes disk texture more conspicuous. Both are artistic visualizations. Neither the screenshots nor the present inspection establish relativistic ray-tracing accuracy, and a brighter or more detailed disk is not itself evidence of greater physical correctness.

6.4 Geometric wormhole exploration

The wormhole case makes geometric structure more directly inspectable than the shader-dominated scene. Both sources construct a two-ended surface using radial profile $r(\ell) = \sqrt{\ell^2 + a^2}$ and axial profile $z(\ell) = a\,\mathrm{asinh}(\ell/a)$, where ℓ is the signed longitudinal parameter and $a > 0$ is the throat radius. Cutaway and grid controls expose the connection between the two mouths and the narrow throat. Figure 19 fixes $a = 1$, enables both controls, and aligns the exterior viewing direction at shader time 6 seconds.

(a) DeepSeek V4.1 Flash 1017 (b) GPT-6 Astra

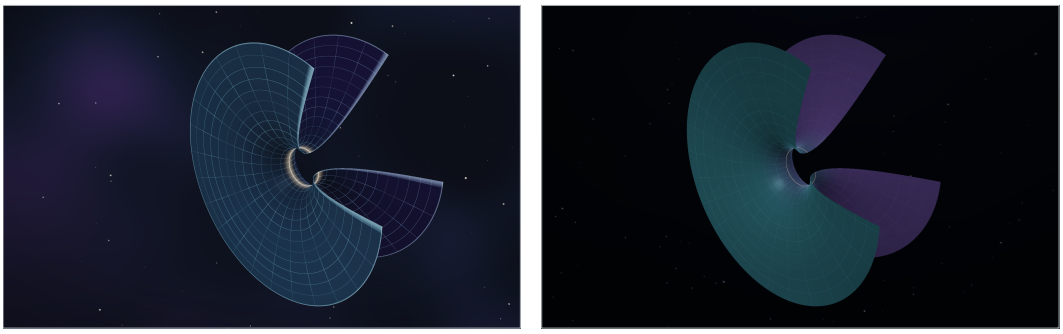


Figure 19: Voder-produced wormhole surfaces at throat radius $a = 1$, with matched exterior viewing direction and enabled grid and cutaway. Native camera fitting and screen-space composition are preserved. Interactive HTML.

The DeepSeek-backed output uses brighter grid lines and stronger cut-edge highlights, making the sampled surface easier to trace against its background. The Astra-backed output uses more subdued shading, with the two sides distinguished mainly by color and the central opening. Both communicate the intended two-ended topology in the sampled exterior view. Their camera journeys are interface features rather than evidence of physical travel through a spacetime, and the shared profile alone does not validate every mesh sample, normal, or transition state.

6.5 Interactive racing and state coherence

The final case adds feedback from user input to the rendered trajectory. Both racing applications contain a procedural circuit, vehicle geometry, a following camera, fixed-step motion updates, and race-state displays. For the visual comparison, we advance each countdown to its native racing state and apply throttle alone for 0.3 seconds in $1/120$-second updates. Figure 20 selects this same early, reproducible instant because both vehicles still occupy similar central screen positions and face forward. No vehicle, camera, geometry, or motion law is altered to manufacture the resemblance. We retain the separate two-second replay solely for the behavioral observation below.

(a) DeepSeek V4.1 Flash 1017 (b) GPT-6 Astra

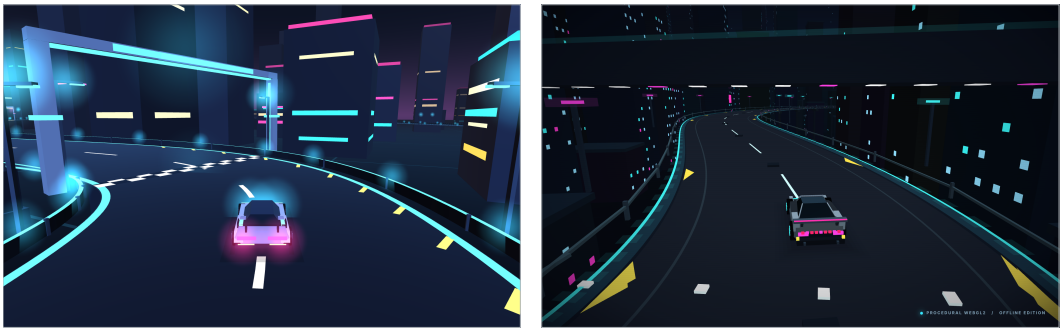


Figure 20: Voder-produced racing outputs after 0.3 seconds of throttle-only input following each native countdown. Both vehicles face forward near the screen center; scene geometry, camera models, speed, and physical trajectories remain native to each application. Interactive HTML.

The DeepSeek-backed output provides a close vehicle view with prominent emissive road furniture, whereas the Astra-backed output provides a wider road view and denser building-window detail. In the separately recorded two-second throttle replay, the former reaches its configured lateral road-boundary clamp while the latter remains

within the visible road corridor; this later observation is not depicted in Figure 20. This distinction concerns the interaction delivered by these two artifacts under one input script. Starting positions, track curvature, acceleration laws, and camera offsets differ, so the observation is neither a normalized driving score nor a ranking of the underlying models. It does show why a convincing still image cannot substitute for checking state evolution.

6.6 Cross-case assessment and scope

Across the collection, Voder delivers runnable applications combining procedural graphics and application-specific state. Their lighting, scene density, and framing differ; the racing replay additionally exposes a behavioral difference. Delivery quality must therefore be inspected separately in rendering, temporal staging, geometric legibility, and input response.

Model attribution and identical prompts within each pair follow the author's execution description and supplied filenames. All nine supplied applications were rendered locally with the same Chromium build and software WebGL renderer. A $1{,}280 \times 800$ CSS-pixel viewport and device scale factor 1.5 produce $1{,}920 \times 1{,}200$ captures. Presentation copies expose existing state-update and drawing functions without changing geometry, shaders, or motion laws. Figures omit interface overlays; paired outputs carry external model labels. Camera directions are explicitly aligned only in the black-hole and wormhole pairs; racing uses native following cameras and identical input time, not forced camera matching. No JavaScript exception or WebGL error was recorded at the sampled states.

These cases demonstrate Voder's use with both model configurations but do not isolate its causal contribution. One artifact per reported condition, without model-only controls, repeated generations, or complete prompt and API traces, cannot support uplift estimates or general model rankings. HTML outputs alone also do not prove execution of the native acceptance path. The case studies complement Section 5 with inspectable deliverables while keeping acceptance guarantees distinct from aesthetic or domain-level correctness.

The collection thus connects capability demonstration to the report's engineering objective: preserve room for creative proposals while keeping acceptance attached to inspectable source and authorized delivery. It does not serve as an ASI capability or general safety evaluation.

7 Limitations and Future Work

The preceding chapters establish a current acceptance architecture and identify the evidence available for it. The remaining work concerns the breadth of specifications, deployment assumptions, interruption coverage, and extension to self-modifying agents. These qualifications define where the control argument applies and what would be needed to strengthen it; they are not replaced by the ASI-agent research objective.

7.1 Extending the substance of acceptance obligations

VGR enforces an externally maintained acceptance boundary, while the configured obligations determine what that acceptance means. The present gateway supports one pinned validator-exit-zero obligation. This provides an executable control point, but finite tests and incomplete specifications can omit behaviors relevant to safety. The boundary can therefore be preserved while its substantive coverage is expanded. Future work will introduce versioned, multi-obligation contracts combining independent regression tests, explicit capability restrictions, adversarial falsification, and proof-carrying checks for tractable properties [10, 16]. A schema change must establish a new baseline and cannot retroactively reinterpret an earlier acceptance.

The research challenge is to turn broader human-defined requirements into checks with declared scope and uncertainty. Undecidable or insufficiently specified properties cannot be made decidable by adding a gate. Such cases require constrained effect scope, additional evidence, or human escalation rather than an automatic success label. This is a boundary on what the available specification and verifier can justify, and a reason to maintain external acceptance authority as those components improve.

7.2 Deployment integrity and effect-specific mediation

Broader obligations must be enforced wherever their relevant effects occur. The current control path assumes trusted host configuration, native binaries, validators, operating-system isolation, and persistence. These dependencies are ordinary deployment requirements, not facts established merely by the use of Rust or hashes. Future work should reduce the privileged interface, strengthen reproducible builds and binary provenance, and add independent integrity witnesses. Remote attestation may support selected deployments, but cannot alone establish semantic correctness.

Extending Voder beyond its software delivery boundary also requires effect-specific mediation [9]. Candidate evaluation must remain confined, and network services, credentials, deployment operations, and physical interfaces need their own authorization and observation contracts. A local restoration protocol cannot reverse information already disclosed or an external commitment already made. Preventive capability restrictions and explicit escalation are therefore necessary for such integrations. The principle is consistent across environments: prior evidence must govern the exact effect, with authority owned outside the proposal-producing session.

7.3 Interruption coverage and cumulative control evidence

The current recovery record supports the tested SDK fixtures and acknowledged blocked-state daemon restarts. It does not cover every interruption between dispatch, native commitment, and destination acknowledgment. The next experiments will inject failures at journal synchronization, evidence publication, final commitment, independent observation, and each writeback phase. Sequential multi-file replacement also requires testing what outside readers can observe and when restoration is permitted. External-service integrations will need reconciliation or compensating protocols matched to their own failure models.

This evidence should be extended through adaptive control evaluations, including forged success claims, stale or substituted artifacts, ambiguous outcomes, and attempts to weaken acceptance policy [6]. Measurements should report erroneous acceptance, unauthorized effects, restoration conflicts, and review burden separately from task completion. Multi-round assessment must account for correlated failures and evaluation overfitting. Passing a finite suite gives evidence about its cases, not a guarantee for indefinitely many future changes; held-out tasks, changing environments, and independent review remain part of the assurance program.

7.4 Controlled self-modification and invariant preservation

Once those foundations are strengthened, Voder can investigate self-modification without silently transferring governance to the modified agent. The existing `ImprovementCandidates` library quarantines method proposals and can qualify independently evaluated candidates for human review. It neither activates them autonomously nor establishes a deployed recursive improvement lifecycle. A first experimental extension should allow isolated changes to agent tools, skills, and planning methods, evaluate the resulting versions on held-out workloads, and adopt qualified versions through a separate authorization path. Modifications to VGR, its verifier, or its governing policy require requalification rather than ordinary self-approval.

The corresponding formal target is preservation of explicit invariants across accepted changes. Let s_t include the agent version, protected policy, and relevant environment state; I be a specified safety invariant; and $V_C(s_t, a_t, s_{t+1}, e_t)$ be an external predicate binding the action, evidence, and actual resulting state under contract C. If the initial state and every accepted transition satisfy

$$I(s_0), \qquad \forall s, a, s', e: \quad I(s) \wedge V_C(s, a, s', e) \Rightarrow I(s'), \tag{7.1}$$

then every finite accepted trajectory preserves I, provided all relevant transitions are mediated and the assumptions remain valid. Environment changes must be modeled or restricted; a contract update must re-establish the preservation condition. This conditional induction argument is a research objective, not a theorem already proved for the current implementation. It specifies how future work could connect the present delivery property to controlled recursive change.

Capability improvement must be measured alongside that preservation. Governed self-modification should be compared with a frozen agent and evaluation-only adoption under matched models, workloads, and resources. The goal is to determine whether useful competence can improve while externally enforced constraints continue to hold, rather than assuming that performance gains imply stronger control.

7.5 A constructive research vision for safe ASI agents

Demonstrating that joint improvement would provide a stronger basis for expanding autonomy. The long-term vision is to construct increasingly capable ASI agents whose consequential changes remain answerable to human-defined requirements and independently enforced evidence. In that vision, progress in VGR would reduce identifiable loss-of-control pathways by deepening specification coverage, protecting governance integrity, and preserving constraints through self-modification. Attaining ASI and establishing this level of control remain open research objectives.

This is a constructive program: enable systems to exceed current problem-solving limits, contribute original scientific and engineering ideas, and work productively with people while preserving meaningful intervention and acceptance authority. Better control can support confidence in adopting useful capability advances; the acceleration itself must be evaluated rather than presumed. Voder's planned public GitHub release will support inspection and reproduction of the present mechanisms. The intended outcome is sustained human benefit and cooperative development, with each broader assurance claim earned through stronger specifications, implementation evidence, and evaluation.

8 Conclusion

Voder explores a concrete mechanism relevant to safe, verifiable ASI agents: increasing a system's ability to propose useful changes should not automatically increase its authority to accept those changes. The implemented VGR path gives this principle operational form in software work. Exact candidate and source identities, trusted evidence admission, fresh native qualification, independent commitment confirmation, and destination-byte checks constrain the transition from proposal to retained state. Conservative recovery preserves known outcomes and fences uncertainty instead of treating incomplete observation as new execution permission.

The reported results support specific parts of that control argument. The retained engineering record includes 318/318 component checks and 420/420 concurrency executions, negative fixtures for invalid commitment claims and identity mismatches, 11/11 revised SDK recovery assertions, and three acknowledged-state daemon restarts. These observations concern their documented interfaces and failure cases. Context experiments measure estimated payload reduction, benchmark audits measure retained task rewards, and interactive applications demonstrate useful delivered artifacts. Those capability and efficiency results complement the governance evidence without substituting for it.

Future work will deepen VGR's evidence obligations, protect governance integrity, extend effect-specific mediation, and test whether external acceptance authority survives controlled agent self-modification. These studies could establish VGR as one component of a safe ASI-agent architecture; general superintelligence and comprehensive safety remain research objectives. Voder contributes an engineering foundation for increasingly ambitious discovery under human-defined constraints: useful proposals may improve, while their acceptance remains independently justified at the boundary where effects become real.

References

[1] M. R. Morris et al. *Position: Levels of AGI for Operationalizing Progress on the Path to AGI*. ICML, PMLR 235:36308–36321, 2024; arXiv:2311.02462, revised September 2025.

[2] Google DeepMind. *AlphaEvolve: A Gemini-Powered Coding Agent for Designing Advanced Algorithms*. Official research announcement, May 14, 2025.

[3] AlphaEvolve Team (Google DeepMind). *AlphaEvolve: How Our Gemini-Powered Coding Agent Is Scaling Impact across Fields*. Official research update, May 7, 2026.

[4] METR. *Independent Assessment of AI R&D Acceleration at Anthropic in October 2026*. Independent assessment, November 5, 2026. Conclusions are uncertainty-qualified; substantial supporting evidence was redacted.

[5] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané. *Concrete Problems in AI Safety*. Preprint, 2016. arXiv:1606.06565.

[6] R. Greenblatt, B. Shlegeris, K. Sachan, and F. Roger. *AI Control: Improving Safety Despite Intentional Subversion*. ICML, PMLR 235:16295–16336, 2024. arXiv:2312.06942.

[7] D. Hadfield-Menell, A. Dragan, P. Abbeel, and S. Russell. *The Off-Switch Game*. IJCAI, 2017. arXiv:1611.08219.

[8] M. Williams et al. (OpenAI). *How We Detect Frontier Model Misalignment*. Official technical article, October 29, 2026.

[9] J. H. Saltzer and M. D. Schroeder. *The Protection of Information in Computer Systems*. Proceedings of the IEEE 63(9):1278–1308, 1975.

[10] D. Dalrymple et al. *Towards Guaranteed Safe AI: A Framework for Ensuring Robust and Reliable AI Systems*. Preprint, 2024. arXiv:2405.06624.

[11] Google DeepMind. *Gemini 3.8 Flash Model Card*. Published September 2, 2026; accessed November 7, 2026. Provider-reported results under the card's evaluation configurations.

[12] METR. *Task-Completion Time Horizons of Frontier AI Models*. Evaluation methodology and results, accessed November 7, 2026.

[13] J. Schmidhuber. *Gödel Machines: Self-Referential Universal Problem Solvers Making Provably Optimal Self-Improvements*. Technical report IDSIA-19-03, 2003; revised 2006. arXiv:cs/0309048.

[14] J. Zhang, S. Hu, C. Lu, R. Lange, and J. Clune. *Darwin Gödel Machine: Open-Ended Evolution of Self-Improving Agents*. 2025; revised 2026. Preprint. arXiv:2505.22954.

[15] Z. Wei et al. *Self-Evolving Coding Agents via Harness-Policy Optimization*. Preprint, October 2026. arXiv:2610.22791.

[16] G. C. Necula. *Proof-Carrying Code*. POPL, pp. 106–119, 1997.

[17] M. Zechner. *Pi: Coding Agent and Multi-Provider SDK*. Source repository and documentation, accessed November 2026.

[18] C. Chen (OpenAI). *Unlocking the Codex Harness: How We Built the App Server*. Engineering article, February 4, 2026.

[19] R. Lopopolo (OpenAI). *Harness Engineering: Leveraging Codex in an Agent-First World*. Engineering article, February 11, 2026.

[20] N. Bonamy and D. Choi (OpenAI). *Codex as a Platform: Build on the Open Agent Harness*. OpenAI Developers, August 19, 2026.

[21] Anthropic Engineering. *Effective Harnesses for Long-Running Agents*. November 26, 2025.

[22] P. Rajasekaran (Anthropic). *Harness Design for Long-Running Application Development*. March 24, 2026.

[23] Z.ai. *ZCode: Coding Agent Harness*. Official source repository and documentation, accessed November 2026.

[24] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press. *SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering*. 2024. NeurIPS. arXiv:2405.15793.

[25] X. Wang et al. *OpenHands: An Open Platform for AI Software Developers as Generalist Agents*. 2025. ICLR. arXiv:2407.16741.

[26] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang. *Agentless: Demystifying LLM-based Software Engineering Agents*. 2024. Preprint. arXiv:2407.01489.

[27] C. Packer, S. Wooders, K. Lin, V. Fang, S. G. Patil, I. Stoica, and J. E. Gonzalez. *MemGPT: Towards LLMs as Operating Systems*. 2023; revised 2024. Preprint. arXiv:2310.08560.

[28] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang. *Lost in the Middle: How Language Models Use Long Contexts*. 2024. Transactions of the Association for Computational Linguistics. arXiv:2307.03172.

[29] W. T. Piriyakulkij, R. Lawrence, A. Curth, S. Karmalkar, and N. Prasad. *Subagents vs Agent Skills: Executing Reusable Knowledge for Long-Horizon Agentic Tasks*. 2026. Preprint. arXiv:2609.09233.

[30] Han Li, HanHaoNing Li, Ziqian Jiang, and Yiling Lou. *When Sub-Agents Work in Parallel: The Promises and Pitfalls of Dynamic Concurrency in Long-Horizon Coding Tasks*. 2026. Preprint. arXiv:2610.10263.

[31] DeepSeek. *DeepSeek Harness: Developer Preview*. Official product and architecture documentation, accessed November 2026.

[32] DeepSeek. *DeepSeek Harness Source Code, Snapshot fcc0ad462172522aaf5c93a7b2e01f6b9f0b0159*. Version-pinned source, inspected November 2026.

[33] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. *All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications*. OSDI, pp. 433–448, 2014.

[34] SQLite Project. *Atomic Commit in SQLite*. Technical documentation, accessed November 2026; cited for persistence principles, not as a claim that Voder uses SQLite's transaction protocol.

[35] National Institute of Standards and Technology. *Secure Hash Standard (SHS)*. FIPS PUB 180-4, 2015.

[36] OpenAI. *GPT-4*. Official model announcement, March 14, 2023; *GPT-4 Technical Report*, 2023.

[37] T. Genewein et al. *From AGI to ASI*. Google DeepMind research report, 2026; arXiv:2606.12683 (v2, August 2026).

[38] OpenAI. *Introducing GPT-5.3-Codex*. Official model announcement, February 5, 2026. Reports use of early model versions in training diagnostics, deployment, and evaluation.

[39] Anthropic. *Introducing Claude Opus 4.6*. Official model announcement, February 5, 2026. Describes improved coding, debugging, and long-horizon agentic tasks.

[40] Anthropic. *Project Glasswing: Claude Mythos Preview*. Official restricted-access model announcement, April 7, 2026.

[41] Anthropic Institute. *When AI Builds Itself: Our Progress toward Recursive Self-Improvement, and Its Implications*. Research analysis, 2026, accessed November 8, 2026. Distinguishes AI-assisted AI development from unachieved full recursive self-improvement.

A Implementation Traceability and Authority Boundaries

This source-to-role map supplements Sections 3–4. Complete relative paths and SHA-256 digests appear in `supplementary_evidence/source_map.json`; the version `0.1.45` is independently confirmed in both the extension package and release VSIX. A displayed interface cannot grant native execution authority.

Table 10: Implementation evidence map complementing the architecture in Figure 1. Paths are relative to the supplied source archive.

Responsibility	Source-level anchor
VS Code entry and operator interface	`extension.cjs`; `general-entry.cjs`; VSIX manifest (0.1.45)
Model/session host and tool exposure	`pi-host.mjs` (Agent Core)
SDK task control and persistence	`control-plane.mjs` (SDK)
Native broker and independent confirmation	`broker.mjs`; `commit-verifier.mjs`
Rust gateway and sandbox eligibility	`voder-gateway/src/main.rs`; `sandbox_plugin.rs`
Verified source delivery and restoration	`verified-delivery.mjs`; `batch-delivery.mjs`
Original-project publication	`original-project-publication.cjs`

The acceptance chain binds source preimages and candidate identity before isolated checking, admits trusted evidence, commits after native audit, independently observes that commitment, and only then delivers checked bytes. Algorithms 1–4 detail these boundaries. Guarantees remain conditional on the trusted host, native runtime, filesystem, and configured contract (Section 3.2).

B Evaluation Evidence and Reproduction Boundaries

This appendix extends Section 5 with the exact score policy and original supplementary evidence: `supplementary_evidence/chapter_5_data_results.zip` (offline engineering and daemon results) and `supplementary_evidence/benchmark.zip` (four-suite reward audit). Neither archive was altered, and no model or benchmark task was rerun for this manuscript.

Table 11: Full-suite benchmark results from confirmed successes, scored over the complete suite size.

Benchmark (task numbering)	Confirmed pass / full suite	Score (%)
DeepSWE v1.1 (0000–0112)	89 / 113	78.8
Terminal-Bench 3.0 (0001–0074)	31 / 74	41.9
Terminal-Bench 4.0 (0001–0066)	30 / 66	45.5
SWE-Marathon (0001–0020)	9 / 20	45.0

A confirmed pass requires a retained numeric `official_reward`=1 and is counted only once per task. The full suite size is the denominator irrespective of which task records were retained. These local audits are not accepted official leaderboard submissions; SWE-Marathon is a *reference-assisted internal control*. All model-connected experiments in Section 5 were author-reported to use DeepSeek V4.1 Flash 1017; offline engineering tests invoke no model API.

The 89 Node and 229 Rust results count component assertions; the 420 stress results count process executions. Synthetic-context token figures use a source heuristic, not provider billing. SDK recovery is verified within its 11 controlled fixtures; 3/3 real `voderd` restarts confirm post-acknowledgment blocked-state replay, not recovery from in-flight native commitment.

C Interactive Demos and Reproducibility

The nine executable three-dimensional demonstration files underlying the five case studies in Section 6.6 **will be publicly released on GitHub** as part of the VeriLoop team's Voder source release. This is a planned public release; no repository URL is asserted before publication. The original HTML files are preserved in the accompanying source archive and have not been altered for this report. The files listed below identify the exact artifacts supporting the figures and the interactive-behavior discussion in Chapter 6.

Demonstration artifact inventory.

1. **Cycling pelican (DeepSeek V4.1 Flash 1017):** `3D_Modeling/pelican/deepseek-v4.1-flash-1017.html`.
2. **Rocket recovery (DeepSeek V4.1 Flash 1017):** `3D_Modeling/rocket/deepseek-v4.1-flash-1017.html`.
3. **Rocket recovery (GPT-6 Astra):** `3D_Modeling/rocket/gpt-6-astra.html`.
4. **Black-hole visualization (DeepSeek V4.1 Flash 1017):** `3D_Modeling/black-hole/deepseek-v4.1-flash-1017.html`.
5. **Black-hole visualization (GPT-6 Astra):** `3D_Modeling/black-hole/gpt-6-astra.html`.
6. **Wormhole exploration (DeepSeek V4.1 Flash 1017):** `3D_Modeling/wormhole/deepseek-v4.1-flash-1017.html`.
7. **Wormhole exploration (GPT-6 Astra):** `3D_Modeling/wormhole/gpt-6-astra.html`.
8. **Interactive racing (DeepSeek V4.1 Flash 1017):** `3D_Modeling/racing-game/deepseek-v4.1-flash-1017.html`.
9. **Interactive racing (GPT-6 Astra):** `3D_Modeling/racing-game/gpt-6-astra.html`.

The full SHA-256 identities and provenance of these artifacts are recorded in `case_studies/DEMO_ARTIFACT_MANIFEST.json`. Until the GitHub publication, readers who possess the supplementary archive can inspect the HTML files locally through `OPEN_3D_DEMOS.html`. The capture procedures and figure identities are given in `case_studies/capture_evidence.json` and `case_studies/figure_specifications.json`. The racing comparison uses two fresh browser pages with 36 fixed-step throttle updates; the two-second behavioral check is separately recorded. These demonstrations provide inspectable application artifacts, not evidence of universal program correctness, public hosting, or a model-only evaluation.