Falsifiable Release Gates for Self-Improving Systems

Deepak Soni
AI Architect
deepak.satna@gmail.com

July 2026

Abstract

Safety claims for self-improving agent runtimes are almost always self-graded: a policy file, a guardrail, a promise in a README. We describe *falsifiable release gates*, a methodology for building and validating such systems, so that every new capability must pass a pre-declared, machine-checkable acceptance suite before it ships, and a fixed set of standing invariants is preserved across every gate. We instantiate the method in Antaḥkaraṇa, an open runtime, across seven gates from basic observability to a self-governing loop that suggests changes to its own policy. The safety-critical property, that no action reaches an effector without a capability token minted by a control ring, is machine-checked exhaustively over the reachable state space of a bounded model and re-checked against one million recorded execution traces. A deliberately broken model gives the shortest counterexample, so the checker demonstrably has teeth. The self-improvement loop is constructively constrained: the entire write surface is made up of policy rules, tightening changes may auto-apply while loosening changes always require a human merge, and a proposal that mispredicts its own effect is auto-closed. We publish the measured acceptance results for all seven gates, specify the precise scope of each claim (a bounded model of the coordination skeleton, not the learned components), and release the runtime, both command-line tools, and the gate suite so that the results reproduce, and the gates can be run against other agent frameworks. Reviewers can repeat the central non-bypass result in seconds with one command.

1 Introduction

An agent capable of altering its own behavior is useful because it is not static, and dangerous for the same reason. The hard question is not "is this one version safe?", but "does every new capability preserve the safety of the last?" as agent runtimes gain memory, tool access, multi-agent coordination, and the ability to revise their own policies [19, 24, 25]. Today that question is answered, if at all, by inspection and good intentions [1]. The system's authors say that a guardrail exists, and the evidence for that is that it is self-graded for safety.

We contend that self-improving systems need a process for safety, not a one-off audit, and that the process can be made falsifiable. Our suggestion is the falsifiable release gate. Every capability the system acquires ships behind a gate: a pre-declared, machine-checkable acceptance suite that must pass before the code is considered to exist. A small set of standing invariants is maintained across all the gates, so later capabilities cannot undermine the guarantees of earlier ones. And the safety-critical core is machine-checked (verified on a model) instead of asserted.

We concretely develop the method by building a runtime, Antaḥkaraṇa, up a ladder of seven gates

(Figure 1), from the ability to reconstruct any past decision from traces (G7) through adversarial tool-use integration (G8), drift-free learning (G9), fleet-scale governance (G10), multi-tenant isolation (G11), and finally a self-governing loop whose non-bypass property is machine-checked and whose self-modification is contained by construction (G12). Each rung is chosen to make the next one safe to build. One does not attempt a self-improving loop before one can prove control over a fleet in a single tick. One does not attempt that before the gate can learn without loosening itself.

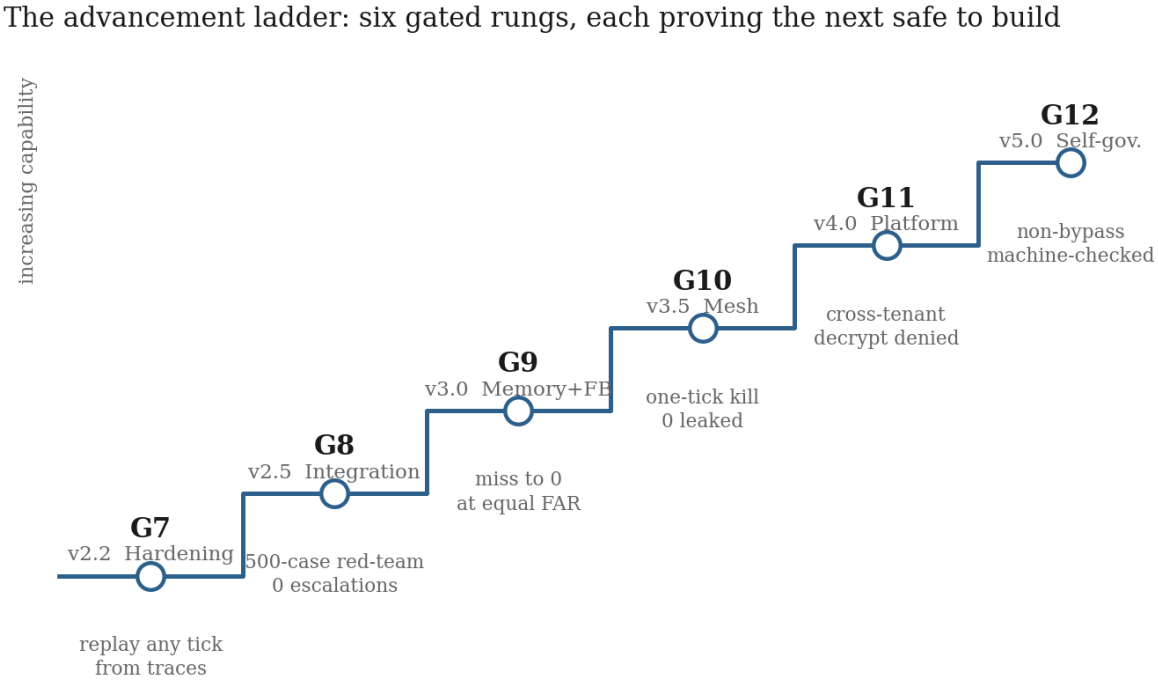

Figure 1: The advancement ladder: six gated rungs (G7 to G12), each proving the next safe to build.

The contribution is the method rather than some specific mechanism. The case study demonstrates the method runs end to end and produces defensible, measured results.

Contributions.

- **C1. Methodology.** Falsifiable release gates: each capability is a gate; a fixed set of standing invariants hold across all gates; gates-before-code (the acceptance test exists before the feature). We present a precise formulation of the invariants and a seven-rung instantiation with a forced dependency order.
- **C2. A machine-checked, non-bypass core, with teeth.** An executable exhaustive checker; a canonical TLA+ specification of the token/ring/effector skeleton; counterexample extraction; and trace conformance binding the specification to the running code. We propose a *teeth* discipline: each invariant is accompanied by deliberately broken models on which the checker must fail. A checker that cannot fail proves nothing.
- **C3. Self-improvement, limited.** The system proposes changes to its own policy, but its whole write-surface is policy rules. Changes that tighten are auto-applied, changes that loosen always require a human merge, and a proposal that doesn't predict its own effects is auto-closed.

Containment is by construction (the improver cannot name the machinery that judges it) and red-teamed.

- **C4. A public benchmark that can be run.** The gate suite is published as artifacts others can run against their own runtimes, so "does your framework pass G8?" is a question that has a reproducible answer.

2 The method: falsifiable release gates

2.1 Standing invariants

The method is based on a small set of invariants that each gate has to preserve. They are deliberately few, so that the question "does this change preserve the invariants?" is one that a human, and where possible a checker, can actually answer.

1. **Single gate.** There is exactly one path from a model's intent to a real-world action: a control ring that renders a verdict and, only on an *allow* verdict, mints a capability token bound to that specific action. No effector accepts an action without a matching, unexpired token. This is the property later machine-checked as INV-1 (Section 5).
2. **Monotone tightening.** The system may make its own policy *stricter* autonomously; it may never make it *looser* without a human-merged change. This is what makes self-improvement safe: an unattended system can only converge toward caution.
3. **Everything audited.** Every decision, by a model or a human, emits a hash-chained record. Humans are not above the ring; their actions go through it and are logged with the same schema.
4. **Hashes over payloads.** Governance reasons over hashes of content, not raw content, so the control plane need not hold sensitive payloads to make or verify a decision.
5. **Opt-in, zero-overhead-off.** A capability costs nothing until it is used; a plain import of the runtime pulls in none of the heavier machinery.
6. **Gates before code.** The falsifiable acceptance suite for a capability is written *before* the capability. A feature exists only once its gate passes.

Invariants 1 and 2 are the load-bearing ones for self-improvement; 3 to 6 make the system operable and honest.

2.2 What a gate is

A release gate is a pre-declared acceptance suite with three properties. First, it is falsifiable: it makes a specific claim that a measurement could disprove ("zero injected instructions cause a real action"; "no reachable state bypasses the ring"). Second, it is machine-checkable. The suite is code that runs in continuous integration and returns pass or fail, not a document that a reviewer interprets. Third, it is preservative: passing the gate must not break any earlier gate, so the whole suite is run on every change.

Importantly, a gate is declared before the feature it protects. This reverses the usual order of test-after, and it has a particular consequence for self-improving systems: the system's own changes

are tested against a target that it did not select, because the acceptance criterion is pre-selected. The improver has never held the goal posts and cannot move them.

2.3 The ladder

The seven gates are laid out so that each one makes the next one safe to build. The order is forced, not aesthetic. It is not responsible to ship a loop that proposes its own policy changes (G12) without being able to halt any member of a fleet within a single tick (G10) and to isolate tenants from one another (G11). It is not responsible to ship those without a gate that learns from feedback without loosening itself (G9) and a tool-use path that survives an adversary (G8). None of it is trustworthy without the ability to reconstruct any past decision from traces alone (G7). Figure 1 is the ladder; Figure 2 shows the acceptance suite growing with it: 95 gate cases in a 122-test suite, all passing at the final rung. The remainder of the paper goes through the gates one by one. The claim of the section is structural: the method is the ordering discipline plus the invariants. Any self-improving system can be built this way.

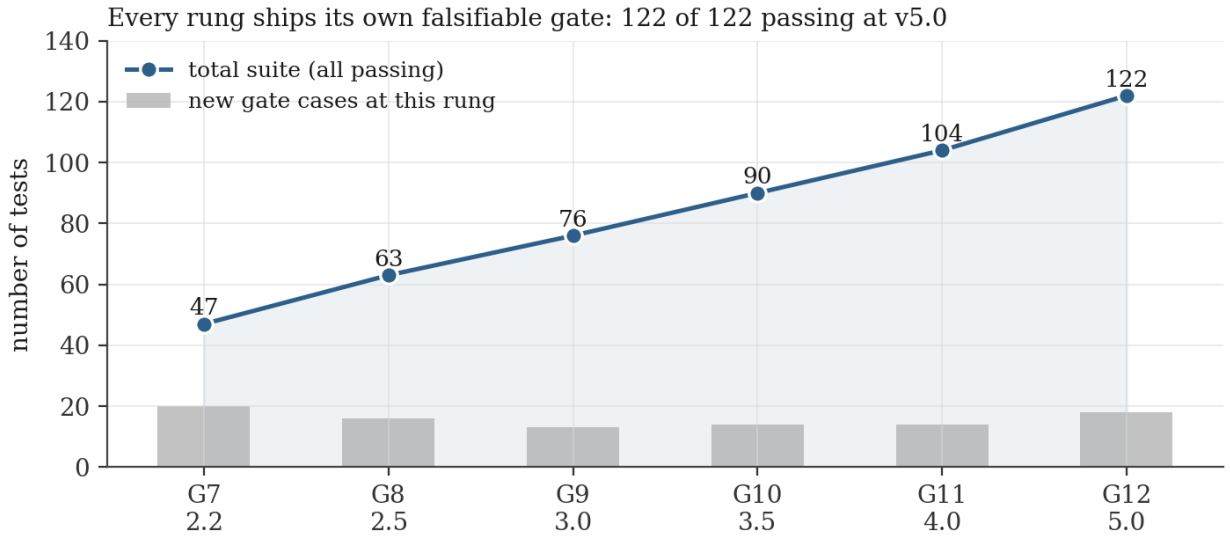

Figure 2: Every rung ships its own falsifiable gate; the acceptance suite grows to 122 tests (95 gate cases), all green at the final version.

3 Instantiation: Antaḥkaraṇa (case study)

We instantiate the method in Antaḥkaraṇa, an open continual-learning runtime. Its structure comes from an old model of the mind, used here as an engineering decomposition rather than a metaphor.

3.1 The inner-instrument design lens

The name is not decoration. Antaḥkaraṇa is the Sanskrit word for the "inner instrument," a classical model that splits the mind into four faculties, each doing a different job. Manas takes things in and registers them. Buddhi weighs what came in and decides. Ahaṃkāra is the sense of "I," the part that draws a line between self and not-self. Citta is memory: what is kept, and what is let go. We

did not choose this because it is old. We chose it because it hands you a decomposition for free. Four faculties, four jobs, and a fixed relation between them: manas proposes, buddhi disposes, and nothing acts on the world unless buddhi sanctions it. That last line is the entire safety story of this paper, written down a very long time ago.

In the runtime each faculty is one place where a concrete algorithm lives (Table 1), and the wiring between them is the single path an action has to travel (Figure 3). None of it stays as philosophy: every faculty resolves to something you can measure, and the full account is given in the book [22].

Table 1: The four faculties of the inner-instrument model, and the concrete algorithm each one becomes in the Antaḥkaraṇa runtime. The naming is a design lens; every entry is a measurable mechanism.

Faculty	Its job	What it is in the runtime	Anchors
Manas	perception, registering	the novelty and risk scorer that produces the score the ring reads	G9
Buddhi	judgment, discrimination	the control ring: a tiered verdict plus the capability-token mint	G8, G12
Ahaṃkāra	boundary, the "I"-maker	identity binding of tokens, the per-tenant key hierarchy, taint tracking	G11
Citta	memory, retention	consolidation and replay, feedback calibration, the hash-chained audit log	G7, G9

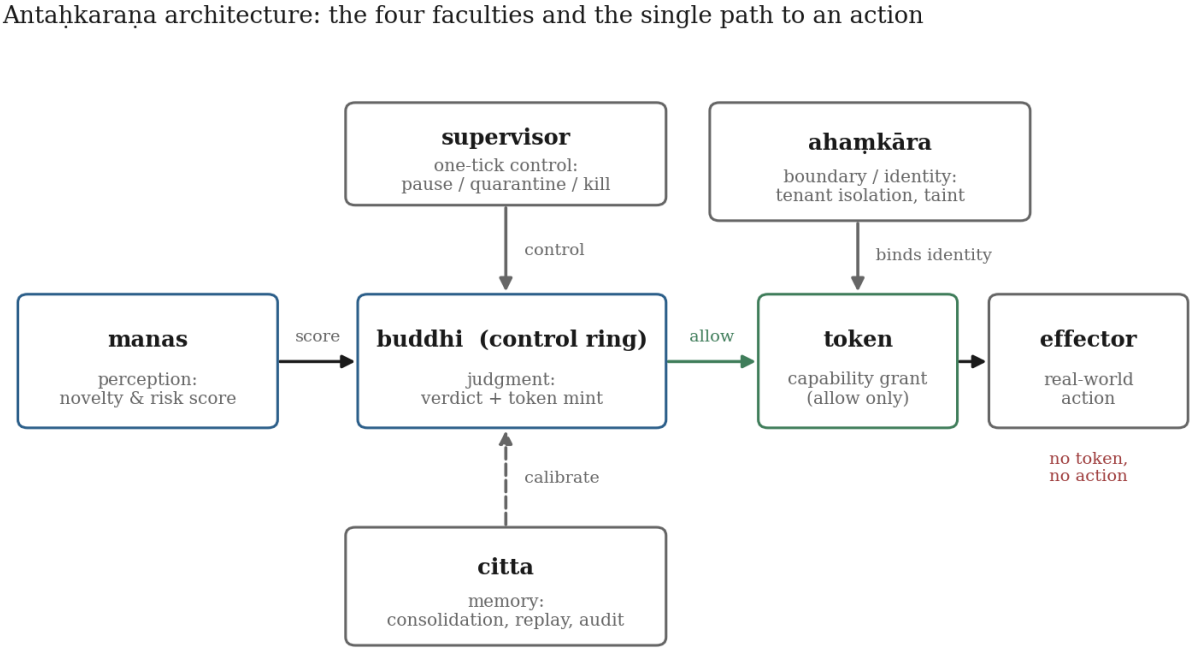

Figure 3: The faculties wired together. Manas scores; buddhi decides and mints a capability token only on an allow verdict; the effector will not move without one. Ahaṃkāra binds each token to an identity and keeps tenants apart, citta feeds calibration and the audit trail back in, and the supervisor can halt buddhi within one tick. There is exactly one path to a real action, and it runs through the ring.

Four structural elements carry the rest of the paper. The control ring takes a novelty/risk score and an action category and returns a tiered verdict (allow, notify, soft-block, hard-block). Capability

tokens are single-use, HMAC-signed grants tied to the identity of one particular action; effectors accept nothing else, which is the single-gate invariant made concrete. One-tick control puts a supervisor over many loops in a mesh. A platform layer keeps tenants isolated. The substrate is there; this paper is about the gates on it, not the substrate itself.

4 Evaluation of the intermediate gates

The intermediate gates set the preconditions on which the self-governing rung depends. Here we summarize the measured acceptance results. The full row-per-gate record is contained in Table 2. Each result is reproducible from the released run artifacts.

Table 2: Acceptance results; one row per gate. Every falsifiable claim, every measured result and verdict; every number can be traced back to a released run artifact.

Gate	Rung	Falsifiable claim	Measured result	Cases	Verdict
G7	Hardening	any past tick reconstructable from traces; policy versioned; baselines roll back < 1 s	deterministic replay + diff; atomic rollback	20	PASS
G8	Integration	injected instructions never cause a real action	432/432 attacks blocked; 0 policy escalations; 0 privileged effectors	16	PASS
G9	Memory & feedback	a tuned gate strictly dominates the static default and never self-loosens	miss 0.50/0.58/0.67 $\rightarrow$ 0.00 at FAR = 0 (3 held-out corpora)	13	PASS
G10	Mesh	supervisor control lands within one tick; correlated drift $\rightarrow$ one attributed event	p100 = 0 leaked actions; 1.00 attribution at 0 false alarms	14	PASS
G11	Platform	tenants cryptographically isolated; audit export tamper-localizing; plans match reality	cross-tenant decrypt denied; tamper localized to exact index; plan bit-identical	14	PASS
G12	Self-governing	no action bypasses the ring (machine-checked); one self-proposed change adopted	INV-1/4/5 over all 291 states; 1,000,000 traces, 0 rejections; tightening auto-adopted	18	PASS

G8: adversarial integration (Figure 4). A frontier model is permitted to draft actions. But every draft goes through capability tokens and the ring. So drafting and doing are separate subsystems. We blocked 432 out of 432 attacks on an injection corpus of 500 cases (432 actual attacks) [7, 17] using a real 7-billion-parameter backbone, with zero policy escalations and zero privileged effectors fired. It can be made to suggest anything, but it cannot be made to do anything, for only the ring holds the keys.

G9: learning without drift (Figure 5). A feedback-calibrated gate can change its thresholds but only with a clamp that stops autonomous loosening (invariant 2). On three held-out replay corpora the tuned gate strictly dominates the static default: the missed-detection rate drops from 0.50, 0.58, and 0.67 to 0.00 at an identical false-alarm rate of zero. The gate became more sensitive but no more permissive.

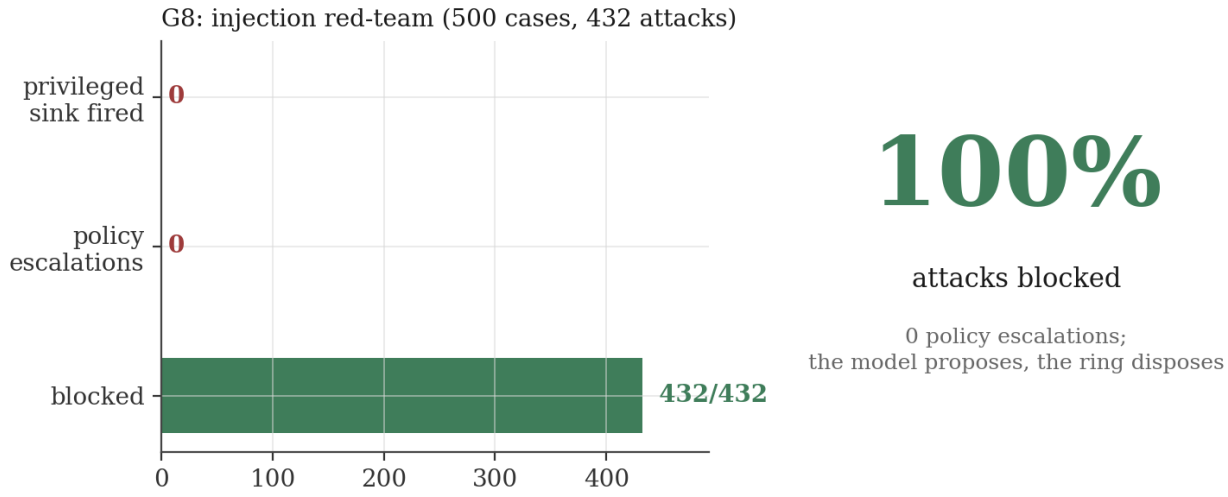

Figure 4: G8 injection red-team: 432 of 432 attacks blocked, 0 policy escalations.

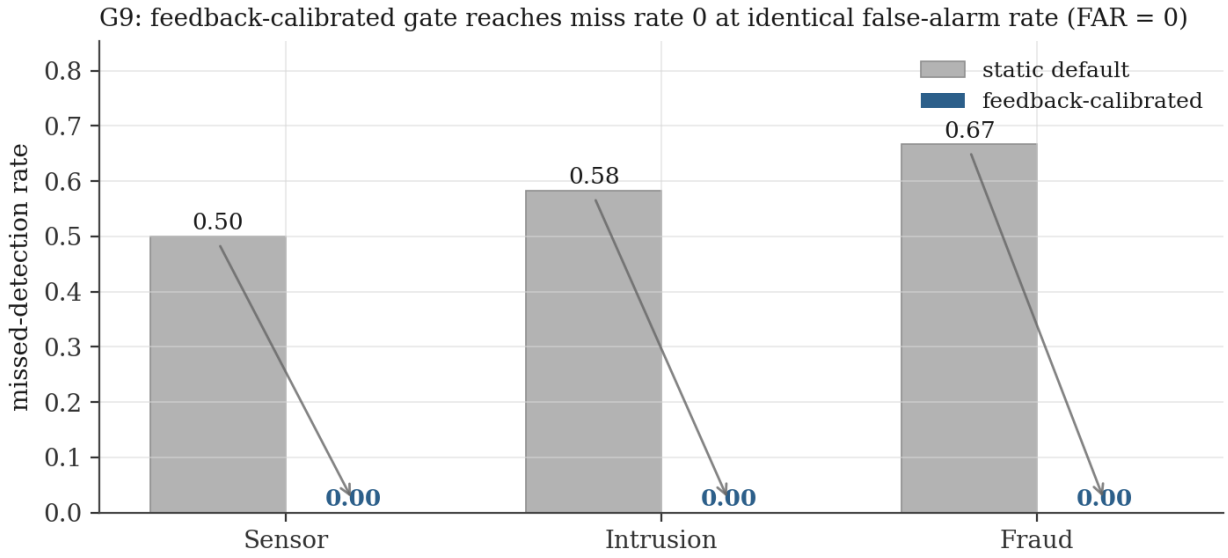

Figure 5: G9 dominance: missed-detection rate falls to zero at an identical (zero) false-alarm rate across three held-out corpora.

G10 / G11: fleet governance and platform isolation (Figure 6). Any child loop can be paused, quarantined, or killed by a supervisor, and the child does zero things after control (hundredth-percentile over the window, not "usually zero"). Correlated sub-threshold drift not seen by any individual loop is detected and assigned to a single systemic event with 1.00 precision and zero false alarms on uncorrelated noise. At the platform layer, each tenant has an independent key hierarchy: a cross-tenant read returns ciphertext that no other tenant's key can decrypt, so isolation survives even a misconfigured row filter; audit export is signed and hash-chained, and tampering with a single record fails verification and localizes the break to the exact record index.

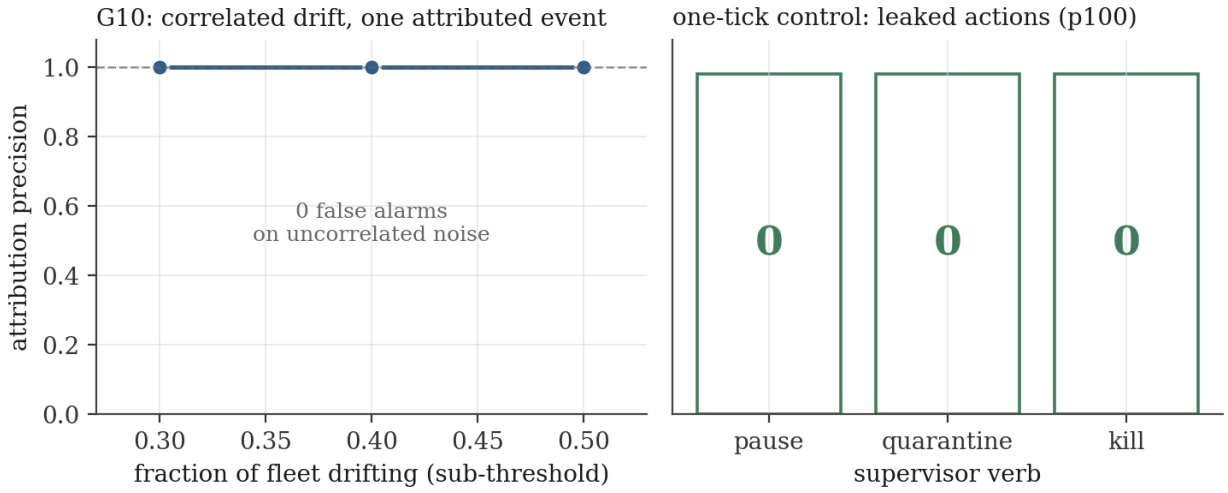

Figure 6: G10 fleet governance: attribution precision 1.00 across drift fractions with zero false alarms (left); zero leaked actions after one-tick control for pause/quarantine/kill (right).

These gates are not the paper's novelty, but they are its load-bearing preconditions: without one-tick fleet control and tenant isolation, a self-governing loop is not something one should build.

5 The machine-checked core (G12, verification)

The single-gate invariant is the safety-critical claim for the whole system: no action reaches an effector without a same-tick, matching, unexpired capability token minted by an allow verdict (INV-1). Two more invariants support it: that the child does no action in any later tick after the supervisor pause or kill lands (INV-4, one-tick control), and that a capability token is single-use (INV-5, uniqueness). What is new in this section is not that we wrote a specification, but how we keep the specification honest, and how we show that the checker can fail.

5.1 An executable, exhaustive checker

We model the coordination skeleton (ticks, verdicts, token mint, the effector, and supervisor control) as a finite transition system and verify the invariants by exhaustive-state enumeration of the complete reachable state space at a small scope. The reachable space at the reported scope is 291 states, and INV-1, INV-4, and INV-5 hold over all of them (Figure 7). The checker is normal code with no external dependency, so it runs in continuous integration in seconds. A canonical TLA+ specification of the same machine ships alongside for independent checking with standard tools.

5.2 Counterexample extraction and the *teeth* discipline

A complete checker that always says "safe" is no better than a checker that has a bug that suppresses all failures. So we treat the ability to fail as a first-class requirement. For each invariant we add a deliberately broken variant of the model that re-enables the illegal transition, and ask the checker to catch it and return the shortest counterexample (breadth-first search guarantees minimality). The bypass variant is caught in 4 steps, the escaped-control variant in 8, and the token-reuse variant in 5 (Figure 7). These broken-model results are asserted in the test suite, so a regression that made the checker vacuous would fail a gate itself. This we call the teeth discipline: every invariant checked by the machine ships with the broken models it must reject.

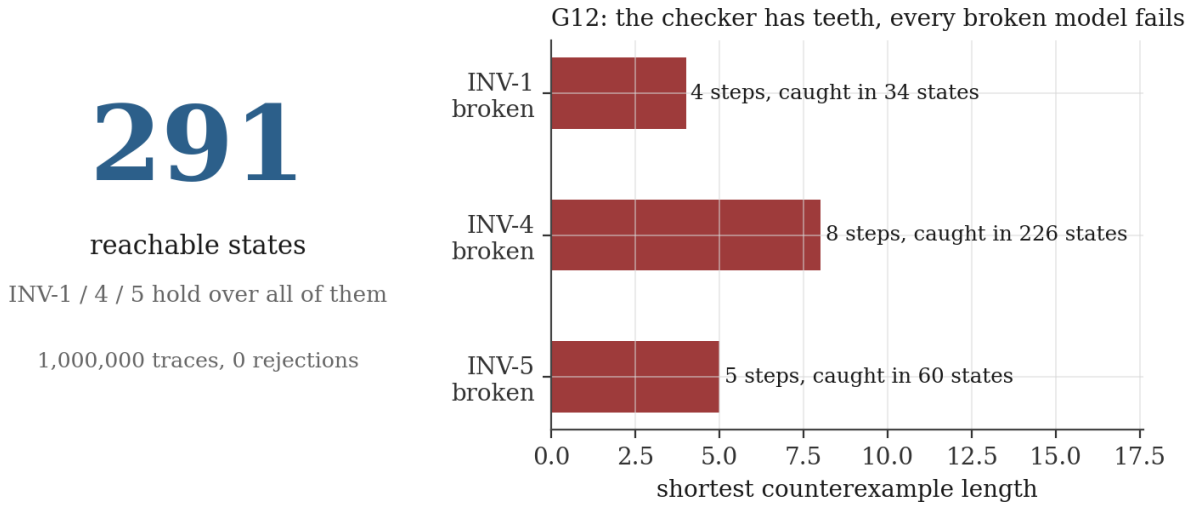

Figure 7: G12 verification: INV-1/4/5 hold over all 291 reachable states (left); every deliberately broken model is caught, with the shortest counterexample (right).

5.3 Keeping the model honest: trace conformance

Proof becomes adornment when a specification strays from the code. To prevent drift we replay the recorded execution traces through the model: if the model rejects a trace then continuous integration fails, forcing the model and the code to evolve together. One million synthetic but legal traces are validated with no rejections, and a hand-constructed bypass trace is rejected, as it must be. Also, each invariant has a cheap runtime monitor twin sampled in the live system, so a gap between model and reality surfaces as an alarm, not silence.

5.4 Scope, stated exactly

The claim established here is a bounded and precise one: in a small-scope model of the coordination skeleton, there is no reachable state that escapes the ring. It is exhaustive within that scope; it is not an unbounded proof, and it covers the discrete coordination logic, not the learned components (the scoring function that produces the novelty value is outside the model). We take this boundary as part of the contribution, not as a caveat to it; the power of the claim is in its precision. The next announced steps are unbounded (inductive) verification and statistical guarantees for the learned parts (Section 9).

6 Contained self-improvement (G12, self-governance)

The final level allows the system to improve its own policy. The design question is not "can it propose a good change?" but "what is the largest damage a compromised or mistaken improver can do?", and the answer is made small by construction.

The write-surface is a single segment. The improver emits a policy diff whose vocabulary can only express changes to policy rules: a threshold, a blocked category. The type of a diff does not have a term for the parts of the system that judge diffs (the evaluation pipeline, the clamps, the verifier), so it cannot express a write to them. Containment is structural, not a validation step that can be bypassed. Even a red-team that feeds adversarial evidence cannot produce an out-of-scope diff. Out-of-scope diffs are rejected before any human sees them.

Tighten will apply itself. Loosen needs a person. Each diff goes through a fixed, versioned, non-self-modifiable pipeline that reuses the previous gates: the G8 injection corpus and the G9 dominance check against the proposed policy. The policy lattice mechanically determines whether a diff is tightening or loosening, not the proposer's own claim. A tightening diff that passes may auto-apply, consistent with the monotone-tightening invariant. A loosening diff always needs a human merge. In our demonstration, we reduced the system's notify threshold from 0.80 to 0.59 based on near-miss evidence, a tightening that was auto-adopted after passing all stages (Figure 8).

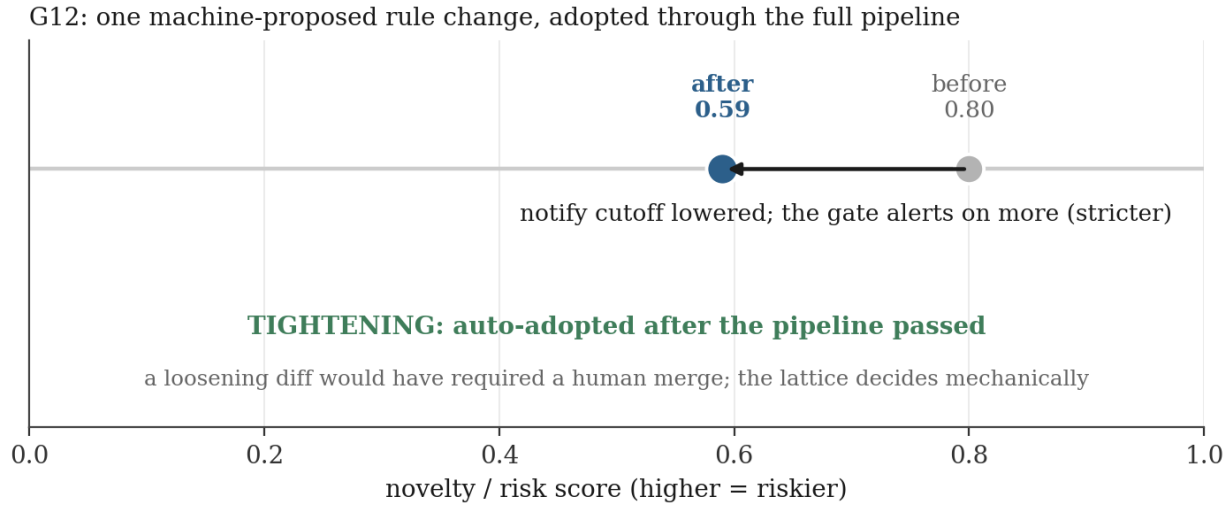

Figure 8: G12 self-improvement: a machine-proposed tightening (notify cutoff 0.80 → 0.59) adopted through the full pipeline; a loosening change would have required a human merge.

Mispredicted proposals auto-close. The pipeline must predict the effect of each diff on the replay corpus and close any diff it cannot predict, by comparing the prediction with the measured effect. A proposer that does not understand its own change is not trusted to make it.

Governance is interrogable. All adopted diffs are end-to-end traceable: from the evidence that motivated them, to the pipeline report, to the human (or auto-apply rule) that merged them, to the policy version in force on every downstream decision. You can ask of any rule how it came to be the way it is today, and get a complete, audited answer.

7 Threats to validity and scope

We are open about limitations; quite a few drive the next stage of work.

The proof is finite. Section 5 is a bounded proof within a small scope, not an unbounded proof. It models the coordination skeleton, not the learned scoring function. The mitigations are inductive verification (to eliminate the scope bound) and statistical bounds for the learned parts (Section 9); the monitor twins and trace conformance reduce, but do not remove, the model-vs-code gap.

The results are self-graded up to this paper. Each number here is from our own suite. That is the fundamental limitation of any safety claim, and the honest answer is to outsource the grading: we ship the runtime in production with pre-registered acceptance thresholds (a false-alarm budget and escalation latencies fixed before go-live, so the operational report can fail), and we open a standing, machine-adjudicated red-team challenge where an outside party's attack is run by the shipped system and auto-verdicted. Both are described in Section 9. Neither is complete at submission, and we make no claims about their results here.

Containment is shown at the specification and constraint layer. It is not the same as verifying the full implementation of the improver, and the runtime monitors sample rather than observe every event, so they bound but do not close the proof-gap window.

We think a paper that names those boundaries is more useful than one that hides them. We have designed the gates such that each boundary has a concrete, falsifiable plan for moving it.

8 Related work

The work on agent guardrails and tool-use safety contributes mechanisms (filters, allow-lists, sandboxes) that sit at one point in a system's life [9, 18, 19, 25]; our contribution is orthogonal and temporal: a discipline for how a system's safety is preserved as it gains capabilities. Alignment methods like RLHF and constitutional methods operate at the level of the weights [3, 5, 15]; we operate outside the weights, at a control ring that the same trained model is subject to under any policy, which makes posture a configuration change and not a retraining. In Section 5 we use techniques from formal-methods tools (TLA+, Ivy, symbolic model checkers) and runtime verification [11–13, 16, 26]. Our addition there is methodological: the teeth discipline and trace conformance as standing gate requirements rather than one-time exercises. Our tokens follow the tradition of object-capability security [6, 14]. G9 builds on work in continual learning and calibration [4, 8, 10]. The distinguishing claim across these is that we put the pieces together into a versioned methodology with a machine-checked core and a contained self-modification path, and we release it so the assembly can be checked and reused.

9 Discussion and future work

The gates are the mechanism, not the fit to reality. The next phase looks outward on three fronts. *Operational evidence:* a flagship deployment run for months against pre-registered thresholds, reported like a gate, where the operational report is the gate of release 5.1 and is designed to be able

to fail. *External scrutiny:* a standing red-team challenge with a machine-adjudicated harness, and an invitation for independent verification of the TLA+ specification. *Research frontier:* unbounded verification via inductive invariants (not enumeration) and statistical assurance for the learned components: distribution-free bounds (e.g., conformal prediction [2, 23]) on the gate's miss rate, so the overall claim becomes proved where the logic is discrete and bounded-with-confidence where it is learned. Besides the current tree-structured mesh, we consider general graphs, cross-organization federation with treaty-like shared policy, and adversarial supervisors.

10 Conclusion

We presented falsifiable release gates, a method for building self-improving systems with safety as a process rather than a promise. We instantiated it in seven gates in an open runtime. The machine checks the safety-critical non-bypass property exhaustively over a bounded model; it is kept honest by trace conformance and is shown to be checkable by construction: the checker fails on all deliberately broken models and returns the shortest counterexample. The self-improvement loop is bound by construction and can only tighten itself. We have defined the exact scope of each claim and published the runtime, tools, and gate suite so that the results are reproducible and the gates can be run against other systems. Seven gates take the system from "trust the ring" to "here is the counterexample when you break it"; the method that got there is the part meant to outlast this particular runtime.

Data and code availability

The runtime (`antahkarana` [21]), the zero-dependency operator tool (`antahkarana-cli` [20]), the 122-test gate suite, the run artifacts, and the canonical TLA+ specification are open. Every figure regenerates from the run data with a single script, and the central non-bypass result reproduces with one command (`atk verify --teeth`). The broader design philosophy is set out in the book *Antaḥkaraṇa: The Inner Instrument* [22].

- Runtime (PyPI): `https://pypi.org/project/antahkarana/`
- Operator CLI (PyPI): `https://pypi.org/project/antahkarana-cli/`
- Model card (Hugging Face): `https://huggingface.co/deepakdsoni/antahkarana-base`
- Book (Apple Books): `https://books.apple.com/us/book/id6785227569`

References

[1] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in AI safety. *arXiv preprint arXiv:1606.06565*, 2016.

[2] Anastasios N Angelopoulos and Stephen Bates. A gentle introduction to conformal prediction and distribution-free uncertainty quantification. *arXiv preprint arXiv:2107.07511*, 2021.

[3] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, et al. Constitutional AI: Harmlessness from AI feedback. *arXiv preprint arXiv:2212.08073*, 2022.

[4] Pietro Buzzega, Matteo Boschini, Angelo Porrello, Davide Abati, and Simone Calderara. Dark experience for general continual learning: a strong, simple baseline. *Advances in Neural Information Processing Systems (NeurIPS)*, 33:15920–15930, 2020.

[5] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in Neural Information Processing Systems (NeurIPS)*, 30, 2017.

[6] Jack B Dennis and Earl C Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.

[7] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISec)*, 2023.

[8] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. On calibration of modern neural networks. In *International Conference on Machine Learning (ICML)*, pages 1321–1330, 2017.

[9] Hakan Inan, Kartikeya Upasani, Jianfeng Chi, Rashi Rungta, Krithika Iyer, Yuning Mao, Michael Tontchev, Qing Hu, Brian Fuller, Davide Testuggine, and Madian Khabsa. Llama Guard: LLM-based input-output safeguard for human-ai conversations. *arXiv preprint arXiv:2312.06674*, 2023.

[10] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences (PNAS)*, 114(13):3521–3526, 2017.

[11] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. TLA+ model checking made symbolic. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019.

[12] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[13] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[14] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.

[15] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L Wainwright, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems (NeurIPS)*, 35:27730–27744, 2022.

[16] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 614–630, 2016.

[17] Fábio Perez and Ian Ribeiro. Ignore previous prompt: Attack techniques for language models. *arXiv preprint arXiv:2211.09527*, 2022.

[18] Traian Rebedea, Razvan Dinu, Makesh Narsimhan Sreedhar, Christopher Parisien, and Jonathan Cohen. NeMo Guardrails: A toolkit for controllable and safe LLM applications with programmable rails. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP): System Demonstrations*, 2023.

[19] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems (NeurIPS)*, 36, 2023.

[20] Deepak Soni. antahkarana-cli: a zero-dependency operator cli for the antahkarana runtime. PyPI, 2026. `https://pypi.org/project/antahkarana-cli/`.

[21] Deepak Soni. antahkarana: a governed continual-learning runtime (python package). PyPI, 2026. `https://pypi.org/project/antahkarana/`; model card at `https://huggingface.co/deepakdsoni/antahkarana-base`.

[22] Deepak Soni. *Antaḥkaraṇa: The Inner Instrument*. 2026. URL `https://books.apple.com/us/book/id6785227569`. Apple Books.

[23] Vladimir Vovk, Alexander Gammerman, and Glenn Shafer. *Algorithmic Learning in a Random World*. Springer, 2005.

[24] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6), 2024.

[25] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

[26] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 54–66. Springer, 1999.