Policy that participates

Kenneth Pernyér, Reflective Labs
Stockholm, Sweden
kenneth.pernyer@reflective.se · reflective.se · ORCID 0009-0006-4543-1156

The natural place to put authorization in a multi-agent system is the kernel, and it is the wrong place. This paper describes Arbiter, which keeps Cedar policy evaluation, delegation verification, budgets and approval gates outside our convergence foundation and registers them as Suggestors — so that policy participates in the loop rather than becoming the loop. The architectural argument is easy: an engine that is also a policy engine has put the component that changes weekly inside the component that must not change. The mathematical argument is the contribution. We show that the confluence result our substrate depends on requires policy to be monotone in the context: a rule that permits an action in a smaller context and denies it in a larger one makes the order in which Suggestors ran observable in the final state, and the unique normal form is lost. We give the two-line counterexample, state the obligation as a property of the policy corpus rather than of the engine, and describe the linter that rejects violating rules. We then connect the assurance ladder to the preceding paper on symbolic search: a runtime decision is ordinary policy provenance, a solver-backed no-violation result is searched evidence, and only a checked artifact would be verification — three tiers, and the discipline is in never promoting one to the next.

1. Introduction

Every system that governs actions eventually acquires an authorization layer, and the layer is almost always put in the middle of the engine. The reasoning is sound as far as it goes: authorization must not be bypassable, the engine is the thing nothing bypasses, so authorization belongs in the engine.

The consequence is a kernel that has to be redeployed when a delegation rule changes. Policy is the fastest-moving part of an organization's rules — a new jurisdiction, a new approval threshold, a reorganization — and the engine is the part whose correctness argument everything else rests on. Fusing them means the part with the most change pressure and the part with the most proof obligation are the same artifact.

Arbiter is the alternative. It owns Cedar parsing and evaluation, policy decisions, delegation verification, and a family of gates — flow, rate limit, budget, approval, data classification, compliance — and it registers all of them as Suggestors in the loop of our substrate (Pernyér, 2025a). The engine still owns the governed promotion path and asks the authority question at the gate; it does not answer it. Policy participates.

This paper is mostly about the price of that arrangement, which turns out to be a property policy must satisfy for the substrate's proofs to survive.

Layer	Responsibility
Converge	Suggestor contract, context, promotion authority, shared gate vocabulary
Arbiter	Cedar wiring, policy decisions, delegation verification, reusable gates
Product	Concrete production policies, rollout, keys, audit retention

Table 1 | The boundary. The rule for where a type belongs: if it is a reusable policy contract for every user of the substrate it moves upstream; if it is a Cedar implementation detail or gate behaviour it stays in the extension.

2. What Arbiter owns

Two elements deserve naming because they are load-bearing later. Delegation tokens are signed with Ed25519, so a delegation is a verifiable artifact rather than a row someone can edit — the chain from an actor to the authority they are exercising is checkable without trusting the store it came from. And a dedicated Suggestor consumes an analysis input and emits an analysis report as *searched evidence*, carrying the execution identity so that a replay can tell whether the result came from an in-process solver lane or an external solver.

3. Monotone policy and confluence

Our substrate proves that a Formation reaches a unique normal form: the run terminates, the step relation is locally confluent, and Newman's lemma gives uniqueness. The local confluence argument has a step that is easy to read past. Two Suggestors eligible at context K must remain eligible after the other has run, *with the same proposals admitted*. Eligibility is monotone because the context only grows. Admission is a different matter, because admission is policy.

> **Definition (Monotone policy).** A policy Π is monotone in the context if, for all contexts $K \sqsubseteq K'$ and all proposals p,
> $\gamma_\Pi(p, K) = \text{promote} \implies \gamma_\Pi(p, K') = \text{promote}.$
> What is permitted on less information stays permitted on more.

> **Proposition (Confluence requires monotonicity).** If Π is not monotone, the step relation need not be locally confluent, and the substrate's unique-normal-form result does not apply.

The counterexample is two lines. Let s_1 propose f_1 and s_2 propose f_2, both admissible at K, and let Π contain a rule that denies f_2 whenever f_1 is present — "no vendor may be selected once a conflict of interest has been recorded", say, written so that it denies rather than escalates. Run s_2 first and both facts are promoted, reaching $K \cup \{f_1, f_2\}$. Run s_1 first and f_2 is refused, reaching $K \cup \{f_1\}$. Both are normal forms and they differ. The order in which two Suggestors happened to run is now visible in what the organization believes.

This is an uncomfortable result, because the offending rule is not exotic. It is the shape of a large fraction of the compliance rules an organization actually has: *once this is known, that is no longer allowed.* Three responses are available, and it is important to be exact about which of them recovers the theorem, because only one does.

Forbid the shape. Reject non-monotone rules outright. This restores confluence, and it is blunt: the underlying requirement is real, and a system that cannot express it will have it implemented somewhere worse.

Escalate instead of denying. Rewrite the rule so that new information triggers an escalation rather than a denial. This does **not** restore confluence — the two orders still reach different states, one having promoted f_2 and one having escalated on it. What it changes is that the difference is now an *exit* rather than a silence: the run ends by telling a human that the answer depended on timing, instead of ending with a context that quietly differs from the one the other ordering would have produced.

Declare the contradiction. Where the rule is really about incompatibility rather than authority, encode it in the contradiction relation instead of in policy. The substrate's consistency invariant then applies: any order that would place f_1 and f_2 in the same context escalates on the pair. The resulting contexts still differ, but the reported outcome — *these two facts cannot coexist* — is identical under both orders, which is the strongest form of order-independence available for a requirement that is itself order-sensitive.

We take the third where the requirement is about incompatibility and the second where it is about authority, and we say clearly that neither recovers the theorem. The honest summary of this section is that **a large class of real compliance rules is incompatible with confluence**, and that the best a substrate can do is refuse to let the incompatibility be silent. The linter rejects non-monotone rules and names the two rewrites in its message; what it buys is not a proof but the certainty that nobody introduced order-dependence without being told.

3.1. The obligation is on the corpus, not the engine

It is worth being precise about where this requirement lives. The engine cannot enforce monotonicity — it would have to quantify over all contexts to check it — and a policy language cannot guarantee it, because the language is expressive on purpose. It is a property of the deployed policy corpus, which means it is a property that must be *tested*, and testing it is exactly a question of the form *can any context exist in which this rule denies what a smaller one permitted?*

That is the shape of question our symbolic assurance extension answers (Pernyér, 2025b), and it is why the first fixture there is an Arbiter invariant. The two extensions were built independently and met at this obligation.

4. Three tiers of assurance

Arbiter's evidence discipline is the same as the assurance extension's, applied to policy.

The position we hold, and which the extension's direction note states, is that we should use more of Cedar before adding a proof assistant. Cedar's validator, runtime regression tests over a fixture matrix, and its symbolic analysis are a long way from exhausted, and its analyzability was a design goal of the language (Cutler et al., 2024) rather than an afterthought. Reaching for Lean or Coq while the policy validator is under-used would be a display of rigour rather than an increase in it.

The assurance lane, in order: state the invariant in a form a domain expert can read; review a fixture matrix for whether the model is adequate to it; validate the policy and schema; run the runtime regression

Tier	What produced it
Policy provenance	A runtime Cedar decision, recorded with the request, the policies consulted and the outcome.
Searched evidence	A solver explored the encoded policy space and found no violation, or found one.
Verification	A proof artifact validated by an independent checker. Not emitted.

Table 2 | The ladder. A runtime decision is ordinary provenance and nothing more: it says what happened once, not what can happen. The discipline is refusing to let a lower tier be recorded as a higher one.

tests; prepare the symbolic analysis; obtain either a counterexample or a no-violation artifact. Each step is cheap relative to the next, and each can fail in a way that makes the next unnecessary.

5. Gates as Suggestors

Beyond the policy decision itself, the extension supplies gates for flow transitions, rate limits, budgets, approvals, data classification and compliance. They are all Suggestors: they read the context, they propose a verdict, and they have no more authority than any other participant.

The uniformity has a practical consequence that took us a while to appreciate. A budget check that is a Suggestor produces a *fact* — "this action would exceed the quarterly budget by so much" — which is in the context, attributable and auditable, whether or not it ends up blocking anything. A budget check that is an `if` statement inside the engine produces a branch. The first can be reasoned about after the fact; the second is only visible if someone logged it.

6. Limitations

Monotonicity is checked by a linter over the corpus, and a linter is a syntactic approximation of a semantic property. Rules that are monotone in every reachable context but not in every representable one will be rejected; rules whose non-monotonicity is hidden behind an indirection may pass. The symbolic lane exists to catch the second kind and is not yet run over the whole corpus on every change.

Delegation tokens are verifiable, and key management is a product concern under Table 1. A verifiable delegation chain rooted in a compromised key is verifiably wrong, and nothing in this extension detects that.

Escalation as the recommended rewrite pushes work onto humans. A corpus that converts every conflict rule into an escalation produces a system that stops frequently, which is safe and can be unusable. We have no principled account of how much escalation an organization can absorb, and we suspect it is the real limit on this design rather than anything mathematical.

Finally, this paper's proposition is about our substrate's confluence result. Systems that do not claim a unique normal form are not affected by it, and we have not investigated what weaker property non-monotone policy is compatible with. The interesting question — whether a confluence-up-to-escalation result holds for corpora that use the second rewrite — is open.

7. Conclusion

Keeping policy out of the kernel is usually argued on modularity grounds and those grounds are fine. The stronger reason is that policy is where an architectural obligation of the whole system turns out to live.

If a Formation is to have a unique normal form, then permission may not shrink as the context grows. That is not a constraint the engine can impose or the policy language can guarantee; it is a property of what an organization actually wrote, and it must be checked mechanically and rewritten where it fails — usually into an escalation, occasionally into a declared contradiction. Getting this wrong does not produce an error. It produces a system whose beliefs depend on the order two Suggestors happened to run in, which is precisely the failure the substrate was built to make impossible.

Policy participates. It does not preside, and it does not hide inside the thing that does.

> **Code and correspondence.** The work described here is implemented, not sketched. The source behind every claim in this paper — the engine, the extension, the fixtures and the tests that hold the invariants — can be shared on request, including the parts that did not work. Write to kenneth.pernyer@reflective.se. We are equally interested in being told where the argument is wrong.

References

Cutler, J.W., Disselkoen, C., Eline, A., He, S., Headley, K., Hicks, M., Hietala, K., Ioannidis, E., Kastner, J., Mamat, A., McAdams, D., McCutchen, M., Rungta, N., Torlak, E., Wells, A., 2024. Cedar: A new language for expressive, fast, safe, and analyzable authorization. Proceedings of the ACM on Programming Languages 8.

Pernyér, K., 2025a. Proposal, gate, commitment: deterministic convergence as a substrate for organizational decisions. Reflective Labs technical report.

Pernyér, K., 2025b. Searched, not verified: bounded symbolic assurance for governed decisions. Reflective Labs technical report.

A. The gate surface, operationally

G1 Decision A policy Suggestor reads a request and proposes a decision with the policies consulted and the outcome attached. The decision is provenance, not promotion.

G2 Delegation Delegation tokens are Ed25519-signed. Verification checks the chain to a root authority; an unverifiable chain is a refusal, not a warning.

G3 Gates Flow, rate-limit, budget, approval, data-classification and compliance gates each propose a verdict fact. A verdict is recorded whether or not it blocks.

G4 Analysis The analysis Suggestor consumes an analysis input and proposes a report as searched evidence, carrying the execution identity so replay can see which solver lane produced it.

G5 Linting The corpus linter rejects rules that are non-monotone in the context, and its message names the two admissible rewrites.

B. The counterexample in full

Let K be a context and s_1, s_2 Suggestors eligible at K, proposing p_1 with fact f_1 and p_2 with fact f_2. Let Π contain

$$\mathrm{permit}(p_2) \text{ unless } f_1 \in K.$$

Order A. Run s_2, then s_1. At K, $f_1 \notin K$, so $\gamma(p_2, K) = \mathrm{promote}$ and the context becomes $K \cup \{f_2\}$. Then s_1 runs and f_1 is promoted, giving $K_A = K \cup \{f_1, f_2\}$. No Suggestor can add anything: K_A is a normal form.

Order B. Run s_1, then s_2. The context becomes $K \cup \{f_1\}$, and now $\gamma(p_2, K \cup \{f_1\}) = \mathrm{block}$. Nothing further can be promoted, so $K_B = K \cup \{f_1\}$ is also a normal form.

$K_A \neq K_B$, so the step relation is not confluent and the substrate's uniqueness result does not apply. Note that both runs are individually correct: each obeyed the policy at every step. What fails is not policy compliance but the existence of a single answer.

Rewrite by escalation. Replace the rule with one that escalates p_2 when f_1 is present. In order A, f_2 is promoted and f_1 follows; in order B, p_2 escalates. The two runs still differ. What has changed is that order B now ends in an exit that names the situation, so the divergence is reported rather than absorbed. The claim for this rewrite is exactly that and no more: it does not restore confluence, it makes the loss of it observable.

Rewrite by contradiction. Declare $f_1 \perp f_2$. Under order A the merge that would introduce f_1 alongside f_2 is refused and the run escalates on the pair; under order B the merge that would introduce f_2 alongside f_1 is refused and the run escalates on the same pair. The contexts differ — one holds f_2, the other f_1 — but the escalation is identical, which is the order-independence that matters to the organization even though it is weaker than the one the theorem asks for.

We note, without having pursued it, that a confluence result *up to escalation* — identifying all states that escalate on the same pair — may well hold for corpora restricted to these two rewrites. Establishing or refuting it is the open question this paper ends on.

