Nick Clark · Published April 25, 2026 · PDF

How to Enforce Compliance Rules on an LLM Output

If you ship an LLM into a regulated setting, you already know the hard part is not generating text but guaranteeing what the model will not say. This guide describes an architecture for enforcing compliance rules structurally, at inference time, as a gate on each candidate transition rather than as a prompt or a post-hoc filter. The approach is disclosed in United States Patent Application 19/647,395 and is presented here as a design you build yourself, not as a shipping library. Its home inventive step is the Inference Control inventive step.

What You Are Building

You are building a compliance layer that decides, for each unit of text your language model wants to produce, whether that unit is allowed to become part of the output. The searcher's problem is specific: you work in a domain with hard output rules (healthcare disclosure limits, financial suitability language, legal privilege, data residency) and you need those rules to hold as a property of the system, not as a hope. "The prompt told it not to" is not an answer an auditor accepts.

The architecture in this guide reframes the question. Instead of asking "how do I catch a bad output after the model makes it," you ask "how do I prevent an inadmissible transition from ever being committed to the output in the first place." The disclosed

design interposes a deterministic gate inside the inference loop. A candidate that violates policy is not emitted, softened, or re-ranked. It is discarded before it can condition anything downstream. That is the property this guide calls deterministic non-emission: because a rejected mutation is never committed, the content it would have produced is never emitted, and the compliance decision is a structural gate rather than a scoring pass.

Who needs this: teams whose exposure is defined by the worst single output, not the average one. If a single non-compliant sentence is a reportable event, average-case guardrails do not describe your risk.

Why the Obvious Approaches Fall Short

There are three common ways teams enforce output rules today, and the filing is precise about why each leaves a structural gap. Described accurately, none of them is worthless; each just fails to give you the guarantee a regulated setting demands.

System-prompt instructions. You tell the model, in natural language, what it may and may not do. This is easy and often helpful, but it is an instruction to a probabilistic process, not a constraint on it. The rule lives in the same space as the content it is supposed to govern, and the model weighs it statistically alongside everything else. It sets a tendency, not a boundary.

Post-generation filtering. You let the model produce a complete output, then run classifiers, toxicity checks, fact-checkers, or human review over the result. The filing identifies the structural problem here directly: post-generation verification operates on the finished output, so it cannot correct problems that are undetectable at the surface, it wastes the compute spent generating text that gets discarded, and it cannot see the intermediate states of the inference process because those states are opaque hidden

activations not exposed to an external evaluator. Worse, the same statistical patterns that produce a policy violation also produce plausible-looking violations that evade the very detector meant to catch them.

Constrained decoding. You mask invalid tokens out of the probability distribution before sampling, forcing valid JSON or syntactically correct code. This is genuinely useful for enforcing output *format*. But, as the filing notes, it operates on individual tokens and masks a probability distribution; it enforces structural validity of form, not semantic compliance with a policy about meaning. Valid JSON can still assert a forbidden claim.

The common gap: each of these treats an inference step as a token selection that can be nudged or cleaned up afterward. The filing's framing is that an inference step is an *execution event* that makes a semantic commitment and conditions every step that follows, and that governing it after the fact is like auditing financial transactions only at year-end.

The Architecture

The disclosed approach, per Chapter 8 of the filing, has four moving parts arranged as a governed inference loop.

1. A semantic state object. Alongside the model's native internal state (its hidden activations and key-value cache, which are numeric and not inspectable), the substrate maintains a separate, structured, typed, inspectable data structure that represents the semantic execution context of the inference run. It is constructed at initialization from the agent's governed fields and the task context; it is not produced by the model. Its schema, as disclosed, includes typed fields for: **intent** (what this inference call is for), **context** (domain, audience, epistemic conditions), **memory** (the semantic commitments established by transitions already admitted), **policy reference** (the governance constraints in force for this run), **mutation descriptor** (the proposed

change a candidate would make), **lineage** (the ordered, timestamped record of admitted transitions and the determination that let each one through), and **entropy and uncertainty bounds** (how much semantic uncertainty is permitted at this step). This object is the reference the gate evaluates against, and because it is inspectable it also becomes your audit trail.

2. Inference transition as semantic mutation. Each candidate the model proposes, whether a token, a multi-token phrase, or a whole reasoning step, is mapped by a mutation mapping module into a structured mutation descriptor before it is judged. The descriptor records which fields the candidate would change, the proposed new values, the semantic category of the change (assertion, qualification, elaboration, negation, reference, transition), and how much novelty it introduces. Transitions that are semantically inert (pure formatting or connective tissue) are classified as inert and passed through without a full evaluation, so the gate does not tax risk-free tokens. That inert/non-inert classification is itself deterministic.

3. The semantic admissibility gate. This is the core. The gate receives a proposed mutation and the current semantic state object and returns exactly one of three outcomes: **admit**, **reject**, or **decompose**. The filing is emphatic that there is no probabilistic scoring, no soft threshold, and no confidence-weighted pass-through, and that the gate is deterministic: the same state plus the same mutation always yields the same determination. It is explicitly not a trained model and not a learned step verifier; it is a deterministic evaluation engine over typed fields whose criteria come from the semantic state object's governance constraints, not from data. A mutation must pass four sequential stages, and failing any one routes it to rejection or decomposition:

- **Policy constraint evaluation** first, because it is the fastest (a bounded comparison) and because policy violations are absolute. The candidate is checked against the policy reference field; a violation is rejected outright.

- **Mutation descriptor validation**, checking the descriptor for internal consistency and consistency with current state (it must not presuppose unestablished content or contradict established content).
- **Lineage continuity validation**, checking that the candidate can be coherently appended to the trajectory of previously admitted transitions rather than representing an unexplained discontinuity or regression.
- **Entropy bounds evaluation**, checking that the introduced uncertainty stays within the permitted bound for this context (tight for high-precision contexts, wider for exploratory ones).

An admitted mutation is committed to the semantic state object and the lineage is extended. A rejected mutation is discarded, and the engine must pick another candidate or terminate; nothing is applied. A decomposed mutation, one too coarse to judge atomically because it bundles admissible and inadmissible changes, is split into sub-mutations that are each submitted to the gate again. Because rejection means "not committed," a policy violation results in the non-emission of that content: the structural gate the lead refers to.

4. Trust-slope continuity validation. The per-step gate cannot, by itself, catch slow drift, where each individual step is locally admissible but the sequence as a whole walks away from the original intent. The disclosed trust-slope mechanism runs across the cumulative lineage, computing a semantic distance for each new admitted transition (content deviation, epistemic-certainty divergence, register divergence). When the trust-slope exceeds a configured threshold it can raise a drift warning, apply a drift correction (re-anchoring context, tightening entropy bounds, narrowing policy), or trigger a drift halt that terminates the run and returns the content admitted before the threshold plus a structured report of where drift was detected. This computation is also deterministic given the same lineage and parameters, and its thresholds live in the policy reference field.

How to Approach the Build

You are implementing this yourself against your own model and runtime. The steps below follow the disclosed structure. Treat the interface sketch as illustrative, not as a package to install.

1. **Get access to candidate transitions.** The gate has to see candidates *before* they are committed. Depending on your stack that means hooking token-level candidate selection, or working at the granularity of reasoning steps in a chain-of-thought or nodes in a tree-of-thought loop. The filing explicitly contemplates all of these granularities. If your only access point is the finished string, you are back to post-hoc filtering and this architecture does not apply.
2. **Define the semantic state object as typed fields.** Model intent, context, memory, policy reference, mutation descriptor, lineage, and entropy bounds explicitly. Keep it inspectable; that property is what makes both governance and audit possible.
3. **Write the mutation mapping.** For each candidate, produce a mutation descriptor (fields touched, proposed values, semantic category, novelty) and a deterministic inert/non-inert classification so formatting tokens skip the full pipeline.
4. **Implement the four gate stages in order**, cheapest and most absolute first. An illustrative sketch:

```
# illustrative only; faithful to the disclosed four-stage order
def admit(mutation, state):
    if violates_policy(mutation, state.policy):        return REJECT
    if not descriptor_consistent(mutation, state):     return REJECT_OR_DI
    if not lineage_continuous(mutation, state.lineage): return DECOMPOSE
    if not within_entropy_bounds(mutation, state):     return REJECT
    return ADMIT
```

The load-bearing property is that this returns one of three discrete outcomes deterministically, with no score you later threshold. Encode your compliance rules as policy-reference checks in stage one.

5. **Wire the loop.** On ADMIT, commit the mutation, extend the lineage, advance the engine. On REJECT, discard and force an alternative candidate or termination. On DECOMPOSE, split and resubmit the sub-mutations. The committed lineage feeds the next candidate's evaluation.
6. **Add trust-slope validation over the lineage** for any long-form or agentic run, with warning, correction, and halt responses and thresholds stored in policy.
7. **Treat the lineage as your compliance record.** Because every admitted transition and its determination are recorded in an inspectable structure, you get the audit trail as a by-product rather than reconstructing it later.

What This Does Not Give You

This is an architecture disclosed in a patent filing, not a product, an SDK, or a benchmarked system. There is nothing to `pip install`. You implement every component yourself, and the correctness of your compliance enforcement is only as good as the policy criteria and mutation mapping you write. The filing describes the mechanism and its determinism; it does not supply performance numbers, and this guide makes none.

Scope boundaries worth stating plainly. The gate's determinism is about the *decision function*: the same state and mutation yield the same admit/reject/decompose outcome. It does not make your model's underlying judgment infallible, and it does not certify factual truth; stage one enforces the policies you encode, no more. If you cannot intercept candidates before commitment, the approach does not apply to your setup, and you are limited to the post-hoc methods it is meant to replace. Constrained decoding remains the right tool for pure output-format validity; this architecture

governs semantic admissibility, which is a different concern. Finally, quality of the entropy bounds, lineage-continuity checks, and trust-slope thresholds is design work you own; the architecture gives you the places to put those rules, not the rules themselves.

Disclosure Scope

The architecture described here (the semantic state object, the mapping of inference transitions to semantic mutations, the deterministic four-stage semantic admissibility gate producing admit, reject, or decompose, and trust-slope continuity validation) is disclosed in United States Patent Application 19/647,395. This guide is educational. It explains an approach a developer can build and reason about; it is not a warranty, a guarantee of compliance or performance, or an offer of software. Implementations, and their correctness under any given regulatory regime, are the responsibility of the party who builds them.

Inference Control (/inference-control) All 40 steps → (/inventive-steps)

Govern inference at the point of generation.

Chapter 8 (/patents/19-647395/chapters/inference)
Explore all disclosures in Inference Control → (/inference-control)

Inference Control overview → (/inference-control)