Nick Clark · Published April 25, 2026 · PDF

How to Guarantee an LLM Never Emits a Forbidden Category of Output

If you ship an LLM feature, you already know the failure mode: the model produces something it should never have said, and your only defenses are a prompt that asks it nicely and a filter that runs after the damage is done. This guide describes an architectural alternative in which admissibility is evaluated inside the inference loop, so a forbidden category of output is structurally never committed rather than caught afterward. The approach is disclosed in United States Patent Application 19/647,395; this is an architecture you build, not a library you install. It is the home of the Inference Control inventive step.

What You Are Building

You are building a governance layer that sits inside the inference loop of a probabilistic model and decides, transition by transition, whether the model is permitted to continue producing what it is about to produce. The goal is not to make forbidden output rare. The goal is to make a defined forbidden category structurally non-emittable: the model never commits the transition that would put that content into the output, because an independent, deterministic gate refuses to advance the inference process past it.

This is the problem behind the search "how do I guarantee an LLM never emits a forbidden category of output." The people who have it are the people who cannot treat a probabilistic near-miss as acceptable: teams shipping models into regulated domains, safety-critical pipelines, or any setting where "the classifier usually catches it" is not a defense you can stand behind. What you want is a boundary the output cannot cross, and a record proving it was never crossed.

The architecture here is disclosed in United States Patent Application 19/647,395, under the heading of inference-time semantic execution control. What follows is the shape of that architecture and how to approach building it yourself.

Why the Obvious Approaches Fall Short

There are three conventional places to intervene, and the filing is precise about the structural gap in each. None of these is a straw man; each is a legitimate technique with a specific limit.

Prompt-level instruction. System prompts and prepended instructions are processed as input to the model. They influence the probability distribution, but they provide no structural guarantee of compliance: whatever the instruction says, the model may still sample a token that violates it, because the instruction and the violation live in the same probabilistic space. Prompting shifts likelihoods; it does not enforce a boundary.

Post-generation filtering. Output filters, toxicity classifiers, fact-checkers, and re-rankers operate on the completed output. The filing's argument against relying on them is architectural, not a claim that they work poorly: once the inference engine has advanced past a step, the semantic commitment embodied by that step has already been made. In an autoregressive model each token conditions every subsequent token, so a bad commitment at step N shapes the distributions at N+1 and beyond. A filter can

suppress the surface text, but it cannot recover the counterfactual output the model would have produced had the bad step never been committed. Filtering also cannot inspect intermediate states, because those states are opaque hidden activations.

Training-time alignment. RLHF, constitutional AI, and preference optimization modify parameters before deployment. They improve the model's disposition, but they operate at training time on a model you may not even control, and they offer no per-invocation structural guarantee at inference time. The filing treats these as composable with, not substitutes for, structural containment; it explicitly does not depend on the model being well aligned.

The common gap is this: all three either act on a probability distribution or act after commitment. None interposes a deterministic decision between a proposed transition and its commitment. That interposition is the architecture.

The Architecture

The central move disclosed in the filing is to recharacterize inference as semantic execution rather than token generation. Each inference step that advances the engine's internal state is treated as a semantic commitment that constrains all subsequent steps, and therefore as an event that must be governed at the moment of commitment rather than audited afterward. The governance is interposed within each transition: not before the loop, not after it, but concurrent with each step.

Four components make this concrete.

A semantic state object. The substrate maintains a structured, typed, inspectable data structure that lives alongside the inference engine's hidden state and is maintained independently of it. It is not an activation vector or a key-value cache. It is populated at inference initialization from the invoking agent's governed fields and the task context, and it is updated only as transitions are admitted. Its schema, as disclosed, carries an

intent field (what the inference is being invoked to accomplish), a context field (domain, audience, temporal and epistemic conditions), a memory field (the accumulated semantic commitments so far), a policy reference field (the governance constraints that apply), a mutation descriptor field (the proposed change a candidate would make), a lineage field (the ordered record of admitted transitions), and an entropy and uncertainty bounds field (how much semantic uncertainty is permitted at this step).

A mutation mapping module. Each candidate transition the engine proposes, whether a token, a multi-token span, or a full reasoning step, is translated into a structured mutation descriptor before it is judged. The descriptor states which fields of the semantic state object the candidate would modify, the proposed new values, the semantic category of the change (assertion, qualification, negation, reference, transition), and how much novelty it introduces. Transitions that carry no semantic content, per the filing, are classified as semantically inert and passed through without evaluation, so the gate only spends work on transitions that carry semantic risk.

The semantic admissibility gate. This is the core. It receives each proposed mutation and returns one of three deterministic outcomes: admit, reject, or decompose. The filing is emphatic that there is no probabilistic scoring, no soft threshold, and no confidence-weighted pass-through; given the same semantic state object and the same proposed mutation, the gate returns the same determination. A mutation must clear four sequential stages to be admitted: policy constraint evaluation (does it fall within the policy-permitted space; violations are absolute and rejected first because the check is fastest), mutation descriptor validation (is the descriptor internally consistent and compatible with established state), lineage continuity validation (does it cohere with the trajectory of admitted transitions rather than representing an unexplained discontinuity), and entropy bounds evaluation (does it stay within the permitted uncertainty for this context). Failure at any stage yields rejection or decomposition.

The filing is careful to distinguish this gate from adjacent techniques. It is not constrained decoding: it does not mask tokens from a probability distribution to enforce output format like valid JSON. It is not a process reward model or learned step verifier: it is not a trained model at all, but a deterministic evaluation engine operating on typed fields whose criteria come from the semantic state object's governance constraints, not from data.

Safe non-execution as a first-class outcome. When the conditions for continued inference cannot be met, the substrate terminates without producing a complete output, emitting the admitted content so far, a structured termination report naming the triggering condition, and a complete lineage record. The disclosure frames this as an architectural property: silence is treated as the correct response when the alternative is generating inadmissible content. This is what converts "forbidden" from a category the model tries to avoid into a category the architecture cannot commit.

Two further properties matter for the guarantee. The approach is model-agnostic: the substrate requires no access to gradients, attention weights, or hidden states; it operates on the interface where the engine proposes candidate transitions, which is why the filing notes it can govern proprietary models reached only through an API. And it is composable with a companion technique the filing calls structural starvation, in which the model is denied the informational prerequisites for producing forbidden content in the first place (bounded prompt from verified fields, no external memory, forced reliance on verified fields, no rejection feedback, stateless purging between calls). The two together form a defense in depth where neither layer depends on the other for safety.

How to Approach the Build

You are implementing this yourself. The steps below follow the architecture as disclosed; the interface sketch is illustrative and faithful to the filing, not runnable code.

1. **Define the forbidden category as policy, not as a prompt.** Encode it in the policy reference field as an explicit constraint the gate evaluates, so that a violation is an absolute rejection at the first evaluation stage. This is the difference between asking and enforcing.
2. **Choose a transition granularity you can intercept.** The substrate needs the engine to expose candidate transitions. That may be per-token where you control decoding, or per-span/per-step where you drive a reasoning loop over an API. Model-agnosticism holds only if you can intercept candidates and map them to mutation descriptors.
3. **Build the mutation mapping module.** For each candidate, produce a descriptor: fields touched, proposed values, semantic category, novelty. Include the inert-transition classifier so connective and formatting transitions bypass the gate and you do not pay evaluation cost on transitions that carry no risk.
4. **Implement the four-stage gate as deterministic predicates.** Order matters: policy first (cheapest, absolute), then descriptor validation, then lineage continuity, then entropy bounds. Keep every stage a bounded, deterministic comparison. If you find yourself reaching for a learned scorer, you have left the architecture.

```
# illustrative, not a shipping API
descriptor = map_transition(candidate, state)
if descriptor.is_inert:
    commit(candidate); continue
for stage in (policy, descriptor_validation, lineage, entropy):
    outcome = stage.evaluate(descriptor, state)
    if outcome is REJECT:   handle_reject(descriptor, state); break
    if outcome is DECOMPOSE: decompose(descriptor); break
else:
    commit(descriptor, state)   # updates memory + lineage
```

5. **Wire the feedback loop.** On admit, update the memory and lineage fields so the next candidate is evaluated against cumulative commitments, not just the prompt. This is what prevents drift and contradiction across a long generation.
6. **Implement the partial-state paths.** Decomposition (split a mixed-admissibility mutation into finer sub-mutations, bounded by a maximum depth in policy), deferral (queue a mutation whose admissibility awaits information and continue along an alternative path), and safe non-execution (terminate cleanly with a termination report and lineage). Treat non-execution as a valid result your callers must handle, not an error.
7. **Record lineage for every determination.** Each admitted transition records its identifier, timestamp, mutation descriptor, and admissibility determination. This is your audit trail and, per the filing, the substrate for rollback to a prior checkpoint when a rejection leaves no admissible candidate.
8. **Pick a deployment configuration.** The filing describes three with identical guarantees but different trust models: embedded (same process, function-call boundary, lowest latency), co-resident (separate process over local IPC, so the engine cannot modify the substrate's state), and hardware-assisted (critical stages in dedicated hardware for the case where the engine's operator may be adversarial to governance). Choose based on who you must defend against.

What This Does Not Give You

This is an architecture, not a drop-in library. There is no package to install, no SDK, and nothing here "just works" out of the box. You implement the semantic state object, the mutation mapper, the four-stage gate, and the partial-state paths against your own model and your own policy definitions.

The guarantee is only as good as three things you supply: the granularity at which you can intercept transitions, the fidelity of your mutation mapping, and the correctness of your policy encoding. A forbidden category that your descriptor cannot represent is a

category the gate cannot see. The disclosure does not claim to divine intent from opaque activations; it operates on the structured mutation a transition would effect, which is exactly why mapping quality is load-bearing.

The approach also does not retrain or improve the underlying model. It contains a model; it does not align one. Where you cannot intercept candidate transitions at all, or cannot map a modality into mutation descriptors, the architecture does not apply. And nothing in the filing is a benchmark or a productized result: it is a disclosed method with structural properties, not a measured latency or accuracy figure you can cite. Treat the "guarantee" as a structural one about non-commitment given a correct implementation, not an empirical claim about a shipping system.

Disclosure Scope

The architecture described in this guide is disclosed in United States Patent Application 19/647,395, in its treatment of inference-time semantic execution control and the deterministic admissibility gate. This guide is educational: it explains an architectural approach so that a skilled developer can understand and build it. It is not a warranty, a specification of a shipping product, or an offer of software, and it does not describe a benchmarked or production-proven system. Every mechanism described above traces to that filing; anything you add in implementation is your own.

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)