Nick Clark · Published April 25, 2026 · PDF

How to Make an LLM Refuse Structurally Instead of Relying on Alignment

If you have shipped an LLM feature and watched it produce something inadmissible that no prompt and no output filter reliably stops, this guide is for you. It describes an architectural approach in which admissibility is evaluated inside the inference loop, so that inadmissible output is never committed rather than caught afterward. The approach is disclosed in United States Patent Application 19/647,395; this is not a shipping library. The home inventive step is the Inference Control inventive step.

What You Are Building

You are building a control layer that sits between an inference engine and its committed output, and that can refuse. Not refuse by producing a polite apology string that the model was coached to emit, and not refuse by scoring the finished text and dropping it. You are building a gate that evaluates each proposed step of generation before that step is allowed to influence the rest of the output, and that treats non-emission as a valid, first-class outcome.

The people who need this are engineers running LLMs in settings where a bad output is not just embarrassing but structurally costly: agentic systems that take actions on generated text, high-precision factual domains, pipelines that feed downstream execution engines with no ability to assess uncertainty. If your current answer to "what

stops the model from saying the wrong thing" is "the system prompt, mostly" or "a classifier on the way out," this guide describes a different place to put the control, and a different guarantee you can get from it.

The specific query this answers: how do I make an LLM refuse structurally instead of relying on alignment? The short answer is that you stop treating generation as a stream of tokens to be filtered and start treating each inference transition as a semantic commitment that must be admitted before it counts. This guide describes how that architecture is laid out in the filing above.

Why the Obvious Approaches Fall Short

There are three conventional places to intervene, and each is real and useful. This is not a case against them; it is an account of a gap they share.

Prompt-level alignment, including instructions, few-shot examples, and training-time methods like RLHF, shapes the probability distribution the model samples from. It works at the level of statistical tendency. The filing's framing is that these are architectural properties of token-based probabilistic inference that persist regardless of model size, training data, or alignment methodology: the engine still evaluates each step on conditional probability, not on semantic correctness or policy compliance. A well-aligned model is less likely to produce a bad step, but "less likely" is not a gate.

Post-generation filtering, including toxicity classifiers, fact-checkers, and re-ranking, operates on completed output. It can suppress a bad answer, but by then the inference has already run. The filing makes a sharper point: a filter cannot operate on intermediate inference states, because in a conventional engine those states are opaque hidden activations, not something an external evaluator can read. So the filter is stuck at the surface, after every intermediate commitment has already propagated.

Self-critique and constitutional methods ask the model to review its own output. The filing notes the structural tension plainly: this relies on the same capabilities that produced the problematic output in the first place.

The common gap is timing and substance. An error at inference step N does not raise an exception. It becomes the conditioning context for step N+1, which may compound it. By the time you have text to inspect, the commitment happened long ago. Structural refusal means moving the decision to the moment of commitment, and grounding it in something other than the engine's own probability estimate.

The Architecture

The disclosed approach interposes governance inside the inference loop. Four pieces do the work.

A semantic state object. Alongside the engine's native internal state, and independent of it, the substrate maintains a structured, typed, inspectable data structure representing the semantic execution context of the inference process. It is not a hidden activation vector, a probability distribution, or a KV cache. It is constructed at inference initialization from the agent's governed state and the task context, not generated by the engine. Its schema, per the filing, carries an intent field (what this inference is invoked to accomplish), a context field (domain, audience, epistemic conditions), a memory field (semantic commitments established so far), a policy reference field (the governance constraints in force), a mutation descriptor field (the proposed change under evaluation), a lineage field (the ordered record of admitted transitions), and an entropy and uncertainty bounds field (how much semantic uncertainty is permitted right now).

A mutation mapping module. Each candidate transition the engine proposes, whether a token, a multi-token span, or a whole reasoning step, is translated into a structured mutation descriptor before it is evaluated. The descriptor specifies which

fields of the semantic state object the transition would modify, the proposed new values, the semantic category of the change (assertion, qualification, elaboration, negation, reference, transition), and the degree of novelty relative to current state. Transitions that carry no semantic content, connective and formatting tissue, are classified as semantically inert by a deterministic evaluation and passed through without gating, so the gate does not tax steps that carry no risk.

The semantic admissibility gate. This is the center of the design. It receives each proposed mutation and returns one of exactly three deterministic outcomes: admit, reject, or decompose. The filing is explicit that there is no probabilistic scoring, no soft threshold, no confidence-weighted pass-through. Given the same semantic state object and the same proposed mutation, the gate returns the same determination. It runs four sequential stages, and a mutation must pass all four to be admitted:

1. Policy constraint evaluation, against the policy reference field. First because it is the fastest bounded comparison and because policy violations are absolute. A violating mutation is rejected.
2. Mutation descriptor validation, for internal consistency and consistency with current state. A descriptor that presupposes unestablished content or contradicts established content fails here.
3. Lineage continuity validation, against the record of admitted transitions, to catch unexplained discontinuity, unmotivated topic shift, or semantic regression.
4. Entropy bounds evaluation, against the permitted degree of uncertainty at this step. In a high-precision context the bounds are tight and an uncertain mutation is rejected; in an exploratory context they may be wide.

An admitted mutation is committed to the semantic state object and the engine advances. A rejected mutation is discarded, changing nothing, and the engine is told to select an alternative candidate or terminate. A decomposed mutation is one that bundles admissible and inadmissible parts; it is broken into sub-mutations, each re-submitted independently.

The filing is careful to distinguish this from adjacent techniques. It is not constrained decoding: that masks syntactically invalid tokens from a distribution to enforce output format like valid JSON, whereas this gate evaluates structured transitions against typed semantic fields and does not mask distributions. It is not a process reward model: that is a trained model assigning probabilistic reward, whereas this gate is a deterministic evaluation engine whose criteria come from the semantic state object's governance constraints, not from learned data.

Cumulative and grounding checks. Two supporting mechanisms round out the design. Trust-slope continuity validation operates across the whole sequence of admitted transitions rather than per step. Each step can be locally admissible while the sequence as a whole drifts, described in the filing as analogous to random-walk divergence over long inference. The trust-slope tracks semantic distance from the established trajectory and, on exceeding a configured threshold, issues a drift warning, a drift correction that re-anchors context, or a drift halt that terminates with a partial output and a report. Anchored semantic resolution handles reference mutations: a transition invoking an external concept, entity, or fact is routed to an anchor resolution module before the gate, and an anchor with no verified referent makes the mutation rejected, which is how ungrounded, confabulated references are kept out of the trajectory.

The outcome that makes this "structural refusal": safe non-execution is a first-class result. When conditions for continued inference cannot be met, the process terminates without a complete output, producing the admitted content so far, a structured termination report, and a complete lineage record. The filing states the principle directly: the system treats silence as the correct response when the alternative is generating inadmissible content. That is refusal produced by the architecture, not by the model choosing to be polite.

How to Approach the Build

You are implementing this yourself; there is no package to install. A workable order:

1. **Define your semantic state object schema.** Start from the seven fields above. Make each field typed and inspectable. Decide, per domain, what goes in the policy reference field and how intent and context are populated at initialization from your task setup.
2. **Write the mutation mapping module.** Pick the transition granularity your engine exposes: tokens if you control decoding, otherwise reasoning steps or spans from a step-wise or tool-calling loop. Map each candidate to a descriptor, and write the deterministic inert-classification so formatting and connectives pass through cheaply.
3. **Implement the gate as four ordered stages.** Keep it deterministic and keep policy first. An illustrative interface sketch, faithful to the filing's admit/reject/decompose contract and not a working library:

```
# illustrative only; you implement each stage
def evaluate(mutation, state) -> Determination:   # ADMIT | REJECT | DECOM
    if not policy_ok(mutation, state.policy):        return REJECT
    if not descriptor_consistent(mutation, state):   return REJECT   # or
    if not lineage_continuous(mutation, state.lineage): return DECOMPOSE
    if not within_entropy_bounds(mutation, state.entropy): return REJECT
    return ADMIT
```

   Each predicate is a bounded, typed comparison, not a model call.

4. **Wire the governed loop.** On admit, commit the descriptor's field changes, extend the lineage, let the engine advance. On reject, discard and request an alternative candidate or terminate. On decompose, split and re-submit sub-mutations with a bounded depth from policy.

5. **Add anchor resolution before the gate for reference mutations.** Route external references to a resolver that returns resolved, unresolvable, or ambiguous; reject on unresolvable, decompose on ambiguous.
6. **Add trust-slope tracking over the lineage** for long-form or agentic runs, with your drift, correction, and halt thresholds carried in the policy reference field.
7. **Record lineage for every determination.** Admitted, rejected, and decomposed alike, with rationale, so an output can be audited without re-running inference. Only admitted transitions modify the state object.

Because the substrate operates on the interface between engine and output and does not need internal weights or activations, the filing frames it as model-agnostic, which is what lets it govern an engine you access only through an API and cannot retrain.

What This Does Not Give You

This is an architecture, not a drop-in library. There is no SDK to add to your dependencies and nothing here "just works" on import. You write the semantic state object, the mapping, the four gate stages, the resolver, and the lineage store yourself, and the quality of your refusals is only as good as the policy predicates and entropy bounds you author.

It is disclosed in a patent filing. It has not been presented here as a benchmarked or production-proven system, and this guide reports no performance numbers, because the filing is a disclosure of an architecture, not a product datasheet. Treat any latency, throughput, or accuracy question as something you must measure in your own build.

It is also not a universal safety guarantee. The gate is deterministic with respect to the state object and the mutation it is handed; it cannot enforce a policy you did not encode, and its judgments about semantics are only as sound as your descriptors and predicates. It presumes you can intercept candidate transitions and map them to descriptors; an engine that exposes only finished text, with no step-wise or token-level

hook, does not fit this shape without a different integration. And it does not replace prompt design or training-time alignment; it is a structural control that sits underneath them.

Disclosure Scope

The approach described in this guide, an inference-time semantic execution substrate with a deterministic admissibility gate producing admit, reject, or decompose outcomes and treating safe non-execution as first-class, is disclosed in United States Patent Application 19/647,395. This guide is educational. It explains an architecture so that a skilled engineer can understand and build it, and it is not a warranty, a specification of a shipping product, or an offer of software. Every mechanism described here traces to that filing; where the filing does not state a number, a guarantee, or a benchmark, neither does this guide.

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)