Nick Clark · Published July 2, 2026 · PDF

How to Constrain LLM Output at Inference Time Instead of With Prompts

If you have tried to enforce policy, tone, or factual discipline on a language model with system prompts and post-hoc filters and watched both leak, the problem is architectural, not a wording problem. This guide describes an approach that moves the constraint inside the inference loop, so an inadmissible step is never committed rather than being cleaned up afterward. The architecture here is disclosed in United States Patent Application 19/647,395 and is named the Inference Control inventive step. It is a design you implement yourself, not a library you install.

What You Are Building

You want a language model whose output obeys hard constraints: stay in a defined policy space, hold a required register and level of definiteness, and never commit a reasoning step that contradicts what it has already established. And you want that to hold even when a user, a jailbreak, or the model's own drift pushes against it.

The obvious tools are a system prompt and a filter on the finished text. Both are soft. A prompt is a request the model can override; a filter runs after the model has already generated. What you are actually building here is different: a control point interposed inside the inference loop, so that every candidate step is evaluated for admissibility

before it is allowed to influence the next step. An inadmissible step is not emitted and then removed. It is never committed in the first place. The result is deterministic non-emission rather than best-effort cleanup.

This is the problem the Inference Control inventive step, disclosed in United States Patent Application 19/647,395, addresses. The sections below trace the architecture to that disclosure so you can build it.

Why the Obvious Approaches Fall Short

Three standard approaches, described accurately:

System prompts and instructions. These condition the model's probability distribution toward desired behavior. They are genuinely useful, but they are soft constraints: the model weighs them against everything else in context, and there is no structural guarantee that any given output honored them. The filing characterizes prompt-level governance as an instruction, not a gate.

Post-generation filters, classifiers, and re-rankers. These evaluate completed output for safety, toxicity, or policy. They can suppress a finished answer, but they run after the inference engine has already advanced. The disclosure's central observation is that this is structurally too late. In an autoregressive model, each step conditions all subsequent steps; a bad commitment at step N shapes the distributions that steps N+1 onward are sampled from. Suppressing the surface text afterward cannot recover the counterfactual output that would have been produced had the bad step never been committed. The error has already propagated.

Constrained decoding. This masks syntactically invalid tokens from the distribution before sampling, which is effective for enforcing output *format* (valid JSON, syntactically correct code). The filing distinguishes its approach from this explicitly:

constrained decoding enforces structural validity of form, not semantic admissibility, and it operates on individual tokens rather than on structured semantic transitions.

The common gap: none of these evaluate the *intermediate* state of the inference process, because in a conventional engine that state is opaque hidden activations (attention weights, key-value caches) with no structured, inspectable representation of intent, policy, or lineage. There is nothing for an external evaluator to check between steps. The architecture below adds that missing structure.

The Architecture

The core move disclosed in the filing is to recharacterize inference as *semantic execution* rather than token generation, and to interpose a governance gate inside the loop. Every candidate step is treated as a semantic commitment that must be admitted before it can influence subsequent steps.

The governed inference loop. The inference engine proposes a candidate transition. That candidate is mapped into a structured *mutation descriptor* describing the semantic change it would make. The descriptor goes to an *admissibility gate*. Only on admission does the change update a *semantic state object*, which then feeds back as the context against which the next candidate is evaluated. Nothing reaches output without passing the gate first.

The semantic state object. This is the structured, typed, inspectable data structure the conventional engine lacks. It is not a hidden activation vector or KV cache; it lives alongside the engine and is maintained independently. The filing gives it a schema of typed fields, including: an *intent* field (what this inference is for, which constrains which candidates are even relevant); a *context* field (domain, audience, temporal and epistemic conditions); a *memory* field (the semantic content established by previously admitted steps, so new steps can be checked against it for contradiction, redundancy, and drift); a *policy reference* field (the governance constraints in force); a *mutation*

descriptor field (the proposed change under evaluation); and a *lineage* field (the ordered record of admitted transitions with their descriptors and admissibility determinations). It is populated at inference initialization from the agent's governed state and the task, and updated after each admitted step.

The admissibility gate: deterministic admit / reject / decompose. This is the heart. Given the same semantic state object and the same proposed mutation, the gate returns the same determination. The filing is explicit that there is no probabilistic scoring, no soft threshold, and no confidence-weighted pass-through; it is a deterministic evaluation engine over typed fields, not a trained model. It runs four sequential stages, and a mutation must pass all four:

1. **Policy constraint evaluation.** Check the mutation against the policy reference field: content-domain restrictions, safety constraints, structural constraints, task-specific constraints. Violations are absolute and rejected. This stage runs first because it is the fastest bounded comparison.
2. **Mutation descriptor validation.** Check the descriptor for internal consistency and consistency with current state: it must not presuppose content that has not been established, contradict established content, or introduce unresolvable dependencies.
3. **Lineage continuity validation.** Check that the mutation can be coherently appended to the existing lineage without an unexplained discontinuity, an unmotivated topic shift, or a semantic regression.
4. **Entropy bounds evaluation.** Check the mutation against the permitted uncertainty bounds for this context. Tight bounds (high-factual-precision contexts) reject an uncertain step; wide bounds (creative or exploratory contexts) may admit it.

The three outcomes: an *admitted* mutation is applied and the lineage extended; a *rejected* mutation is discarded and the engine is told to pick another candidate or terminate; a *decomposed* mutation (one too coarse to judge atomically, bundling

admissible and inadmissible changes) is broken into sub-mutations that are each resubmitted to the gate.

Register and definiteness as a governance input, not a post-filter. The filing discloses an *output register* field: a structured state governing formality, vocabulary, and precision. It is explicitly not a post-processing filter. A candidate transition whose stylistic characteristics are inconsistent with the current register (for example, highly technical vocabulary when the register indicates informal communication) receives a reduced admissibility score and participates in the gate's evaluation of that candidate. So linguistic and register constraints are enforced at the same point, and by the same mechanism, as policy constraints.

Confidence gating and structural non-emission. The substrate tracks a rolling admission rate over a window. When too many proposed transitions are being rejected, the process transitions from executing mode into a non-executing *inquiry* mode: instead of forcing output, it returns structured queries naming the information deficiencies, policy ambiguities, or contextual gaps that would need to be resolved before admissible inference can resume. The filing frames this as a first-class constructive result, not an error. This is where "deterministic non-emission" is realized: when nothing admissible can be committed, the system withholds output by design rather than emitting a low-confidence answer and hoping a downstream filter catches it. As the disclosure puts it, no output survives admissibility evaluation unless it is compatible with the governing state.

How to Approach the Build

Concrete, ordered steps to implement this architecture yourself. The interface sketches below are illustrative and faithful to the disclosed structure; they are not shipping code.

1. **Define your semantic state object schema.** Start from the disclosed fields: intent, context, memory, policy reference, mutation descriptor, lineage, and (if you need it) entropy bounds and output register. Make it typed and inspectable. This object is the reference your gate evaluates against, so its fidelity determines your ceiling.
2. **Decide your transition granularity.** The filing notes a candidate transition may correspond to a single token, a multi-token phrase, or a complete reasoning step. Coarser transitions (a sentence, a tool call, a chain-of-thought step) are cheaper to gate and align better with semantic evaluation than per-token gating. Choose the level at which "semantic commitment" is meaningful for your task.
3. **Build the mutation mapping layer.** Translate each raw candidate from the engine into a structured mutation descriptor: which fields of the state object it would change, the proposed new values, and the semantic relationship to current values. This is what makes the opaque generation step inspectable.
4. **Implement the four-stage gate as pure, deterministic functions.**

```
# illustrative sketch, faithful to the disclosed four stages
def evaluate(mutation, state):
    if not policy_ok(mutation, state.policy):       # stage 1: absolute
        return REJECT
    if not descriptor_consistent(mutation, state):  # stage 2
        return REJECT_or_DECOMPOSE
    if not lineage_continuous(mutation, state.lineage):  # stage 3
        return DECOMPOSE            # restore continuity via sub-steps
    if not within_entropy_bounds(mutation, state.context):  # stage 4
        return REJECT
    return ADMIT
```

Keep it deterministic: same state plus same mutation must always yield the same outcome. Resist the urge to make any stage a learned classifier; the disclosed property is that the gate is a rule evaluation over typed fields, which is what gives it repeatability and auditability.

5. **Close the loop.** On ADMIT, apply the descriptor, extend lineage, and feed the updated state back as context for the next candidate. On REJECT, request an alternative candidate or terminate. On DECOMPOSE, split into sub-mutations and resubmit each.
6. **Add the confidence gate.** Maintain a rolling admission rate. Below your threshold, switch to inquiry mode and emit structured questions about what is missing, rather than forcing an answer. This is your non-emission path; treat its output as a valid result surface in your API, not an exception.
7. **Encode register and definiteness as policy inputs.** Put your required register and precision into the state object and evaluate candidates against them inside the gate, so tone and definiteness are enforced structurally rather than requested in a prompt.
8. **Log the lineage.** Because every admission carries its descriptor and determination, you get a per-step record of why each committed step was allowed. Persist it; it is your audit trail.

What This Does Not Give You

This is an architecture, not a drop-in library. There is no package to install and nothing here "just works" out of the box; you implement the state object, the mapping layer, and the gate against your own engine and policies, and the quality of your result depends entirely on how well you specify them.

It is disclosed in a patent filing. It is not a benchmarked or productized system, and this guide reports no performance numbers, because the filing states none to report. Do not represent it as shipping software.

It also does not replace training-time alignment or downstream monitoring; it is a runtime control point, and it is only as good as the policy and consistency rules you encode. A gate cannot enforce a distinction you never defined. The approach fits tasks where semantic commitments are meaningful and inspectable and where withholding output is acceptable when nothing admissible can be committed. Where you genuinely cannot afford non-emission, or where per-step gating cost is prohibitive, the tradeoff shifts and you should weigh it deliberately.

Disclosure Scope

The approach 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 architecture so a developer can understand and build it themselves. It is not a warranty, a specification, or an offer of software, and nothing here should be read as a representation that a productized or benchmarked implementation exists.

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)