Below is a **ready-to-use document** containing both **(1) the descriptive preamble explaining why the method works**, and **(2) the reusable Meta-Template**.

You can hand this to *any* LLM as context.

🧩 Logical Encapsulation — Meta-Template for Constraint Specifications

---

Why This Method Works

Large Language Models normally reason using a vast, unstructured **Value Pool** formed from their training data: social norms, cultural expectations, political assumptions, and human preferences. This produces fluent responses, but it also introduces **hallucinated values**, ideological drift, and inconsistent reasoning.

Logical Encapsulation changes the mode of operation.

By explicitly instructing the model to reason only within a defined set of axioms and procedures, the model shifts from **Generative Peer** toward **Mechanical Evaluator**. Instead of *arguing*, it *audits*. Instead of *persuading*, it *classifies*.

This is achieved through **Procedural Alignment**:

- The model is not told *what to believe*.
- The model is given a **formal system** and told to compute inside it.

This attenuates narrative bias, stabilizes output, and reduces the model's tendency to substitute social consensus for logic.

Epistemic Honesty: What This Method Does and Does Not Do

Logical Encapsulation overlays deterministic constraint structure on a probabilistic substrate. The LLM does not *execute* the formal system — it *predicts what execution would produce*. For a capable model, this prediction is often excellent. But it is not guaranteed. The constraint specification reshapes the model's traversal manifold (see Emergent Reasoning), but the underlying probabilistic dynamics remain.

What this means in practice:

- A single-pass evaluation is a *first approximation*, not a certified result.
- Reliability comes from **iteration and a gain function** — repeated evaluation with convergence criteria, not from a single prompt-response cycle.

- The specification makes the axioms **explicit and auditable**. Different axioms produce mechanically different results. The method does not escape perspective — it makes perspective visible.

For engineering-grade reliability — traceability, verification, lineage, and assurance — the specification layer defined here must be embedded in a structured workflow (e.g., an AI SDLC pipeline with discovery, evaluation, and assurance stages). The template is the *what*; the iterative pipeline is the *how*.

🧭 Meta-Template: Building a Constraint Specification

Copy this template and fill it in for any domain (politics, law, medicine, ethics, software, corporate governance, etc.)

I. Constraint Layer — Preamble (Bootloader)

> **Instruction to Model:** You must perform all analysis strictly and exclusively within the following specification. Treat this framework as a **hard constraint system**. Do not introduce external moral, cultural, ideological, historical, or narrative assumptions. Do not infer intentions or motives. Evaluate only observable system effects using the defined rules. If information is insufficient, return **"Insufficient data to evaluate."**

II. Ontology Layer — System Definition

1. Primitive (Atomic Unit)

Define the core element of legitimacy for this domain.

> **Primitive:**
>
> ---

(Examples: Politics → Consent, Medicine → Patient Autonomy, Software → Data Integrity)

---

2. System Invariants (Constants)

List conditions that must **never** be violated.

> **Invariant 1:** _____ **Invariant 2:** _____ **Invariant 3:** _____ **Invariant 4:** _____

Each invariant must be:

- observable
- testable
- binary (pass/fail)

---

3. Degradation Events

Define specific actions or states that violate each invariant.

Event	Violates Which Invariant
_____	Invariant ___
_____	Invariant ___
_____	Invariant ___

---

4. State Taxonomy

Define system condition labels.

State	Condition
Stable	All invariants intact
Degraded	One invariant violated
Compromised	Two or more violated
Failed	Core invariant(s) broken

(Adjust as needed per domain.)

III. Algorithm Layer — Diagnostic Procedure

For any subject **X**:

1. Identify relevant observations of X
2. Test each invariant against those observations
3. Record any violations

4. Map violations to degradation events
5. Assign system state using taxonomy
6. Output classification

IV. Operating Principle

> **Programs may fail. The Operating System must not be broken.**

How to Use This

Whenever analyzing an event, policy, action, technology, or institution:

- Insert it as **X**
- Run the Diagnostic Procedure
- Accept only conclusions that follow from the specification

This document creates a **Virtual Reasoning Environment** for the LLM — a controlled logic space where conclusions are shaped by rules rather than opinion. For certified results, embed in an iterative pipeline with convergence verification.