Operational Safety via Recursive Constraint Satisfaction:

A Neuro-Symbolic Architecture for Critical Infrastructure

Stefan Paetzold
CausaNova Research

January 2026

Abstract

Abstract. Large Language Models (LLMs) operate stochastically ($P(\text{next\_token})$), making them inherently unreliable for safety-critical domains such as industrial automation (IEC 61131-3) and public administration. Current alignment techniques reduce the probability of errors but cannot enforce hard constraints. This paper introduces **CausaNova**, a neuro-symbolic architecture that decouples *planning* (Neural) from *execution* (Symbolic).

Unlike static code generators, CausaNova utilizes a **Self-Extending Meta-DSL** (Domain Specific Language) rooted in JSON. This allows the system to not only execute plans but to autonomously expand its own logical vocabulary at runtime. By implementing a **Guard Resolver** utilizing Satisfiability Modulo Theories (SMT), we ensure that this recursive self-modification adheres to strict operational limits. We demonstrate that "Operational Alignment" can be reduced to a dynamic constraint satisfaction problem, effectively eliminating execution-layer hallucinations.

1 Introduction: The Stochasticity Gap

The integration of Generative AI into deterministic environments faces a fundamental contradiction, which we term the **Stochasticity Gap**. Industrial control systems and legal frameworks require a reliability of $\approx 99.999\%$, whereas State-of-the-Art (SOTA) LLMs operate probabilistically.

We argue that *Safety* cannot be achieved through model training (RLHF) alone. Instead, we propose an architectural solution where the LLM acts as a heuristic proposer within a deterministic "exoskeleton"—the CausaNova Engine.

2 Core Architecture: The Neuro-Symbolic Loop

Our architecture implements a "Plan-Validate-Execute" cycle. This cycle is mediated by a formal solver and operates on a structured intermediate representation.

2.1 Layer 1: The Heuristic Proposer (Neuro)

The Neural Layer (e.g., Qwen 2.5) analyzes the user intent. Crucially, it does **not** generate executable code directly. It generates an **Abstract Intent** represented in the CausaNova JSON-DSL.

$$f_{LLM}(\text{Prompt}, \mathcal{S}_{current}) \rightarrow \mathcal{P}_{proposed}$$

Where $\mathcal{S}_{current}$ represents the current state of the Domain Schema (the vocabulary available to the AI).

2.2 Layer 2: The Guard Resolver (Symbolic)

The core innovation is the **Guard Resolver**. It validates $\mathcal{P}_{proposed}$ against a set of constraints $\mathcal{C}$. Since real-world constraints involve arithmetic and temporal logic, we employ **Satisfiability Modulo Theories (SMT)**.

Let $\mathcal{C}$ be the conjunction of constraints derived from Physics ($\mathcal{C}_{phy}$), Law ($\mathcal{C}_{law}$), and Schema Definitions ($\mathcal{C}_{schema}$). The validation function V is defined as:

$$V(\mathcal{P}, \mathcal{C}) = \begin{cases} \text{SAT} & \text{if } \mathcal{P} \models \mathcal{C} \\ \text{UNSAT} & \text{otherwise} \end{cases} \tag{1}$$

2.3 Layer 3: Closed-Loop Error Correction

If $V(\mathcal{P}) \to$ UNSAT, the specific violation (the "Unsat Core") is fed back to the LLM. To prevent the Halting Problem, we introduce a strict iteration bound k_{max}.

3 The Meta-DSL: A Living Ontology

Most "No-Code" tools rely on static configuration files. CausaNova utilizes a **Meta-DSL**—a recursive JSON structure that defines not just the *state* of the application, but its *logic*, *data models*, and *interface components*.

3.1 Structure of the DSL

The DSL is domain-agnostic. It abstracts technical implementation details into logical primitives. A typical component definition in the DSL looks as follows:

```
{
  "component": "bio.analysis.blood_sugar",
  "type": "decimal",
  "constraints": {
    "min": 0,
    "max": 1000,
    "unit": "mg/dL"
  },
  "logic": "if (value > 140) return 'hyperglycemia';",
  "audit": true,
  "ai_risk": "mid"
}
```

Listing 1: Self-Contained DSL Component Example

This single JSON object is compiled by the engine into three distinct artifacts:

1. **Persistence Layer:** The engine automatically migrates the PostgreSQL schema (`ALTER TABLE patients ADD COLUMN blood_sugar DECIMAL(10,2);`).
2. **Interface Layer:** It renders a validated HTML5 input field or API endpoint.
3. **Governance Layer:** The Guard Resolver ingests the constraints (`max: 1000`) and enforces them on every future transaction, regardless of the source (User or AI).

3.2 Decoupling Logic from Runtime

Because the entire application logic is stored as structured data (JSON) and not compiled binary code, the "Brain" (LLM) can manipulate the software architecture as easily as it manipulates text. This is the prerequisite for the system's self-learning capability.

4 Recursive Extensibility (Self-Learning)

A static system cannot adapt to new problems. CausaNova implements a **Recursive Schema Expansion** mechanism, allowing the system to learn new concepts at runtime without recompilation.

4.1 The Learning Cycle

When the Neural Layer encounters a problem that cannot be solved with the existing DSL vocabulary $\mathcal{S}_{current}$:

1. **Detection:** The LLM signals a "Vocabulary Gap" (e.g., "I need to calculate a trajectory, but I lack a physics vector component").
2. **Proposal:** The LLM generates a *Schema Definition* for a new component (e.g., `physics.vector3`) including its constraints and behavior.
3. **Meta-Validation:** The Guard Resolver validates the *definition itself*.
   - Does the new component violate system stability?
   - Are the data types valid?
   - Is the execution logic within safe bounds (e.g., no infinite loops)?
4. **Hot-Integration:** If valid (SAT), the new definition is merged into the Meta-DSL.
5. **Execution:** The system immediately uses the new component to solve the original problem.

$$\mathcal{S}_{new} = \mathcal{S}_{current} \cup \{\text{NewComponent}\} \quad \text{iff} \quad V(\text{NewComponent}, \mathcal{C}_{meta}) == \text{SAT}$$

This mechanism allows CausaNova to evolve from a generic engine into a domain-expert simply by accumulating validated knowledge in its DSL.

5 System Architecture & Implementation

The theoretical model is backed by a production-ready implementation designed for scalability and isolation.

- **Kernel:** Built on **ASP.NET Core 8**, ensuring high-performance request handling and type safety.
- **Solver Isolation:** The numerical solving capabilities are offloaded to isolated Python kernels via a **Task Queue** pattern. This ensures that long-running calculations do not block the governance loop.
- **Infrastructure as Code:** The system utilizes a Kubernetes Service layer to autonomously provision Ingress routes and SSL certificates for generated tenants, demonstrating "Self-Hosting" capabilities.

6 Empirical Validation

6.1 Case A: Mathematical Rigor (FrontierMath Context)

We evaluated the architecture on optimization problems similar to FrontierMath Tier 4 (e.g., BMO space integration).

- **Process:** The LLM did not try to "guess" the number. It used the DSL to construct a solver pipeline using Python kernels.
- **Result:** While the LLM cannot "solve" the math, it successfully "orchestrated" the solver. Correctness depends on the solver's precision, not the LLM's weights.

6.2 Case B: Regulatory Compliance (GovTech)

Applied to German "Elterngeld" (Parental Allowance) applications.

- **Performance:** SMT validation added < 5ms overhead.
- **Safety:** The system rejected 100% of plans that violated hard-coded legal upper bounds.
- **Scale:** Successfully generated 2000+ application forms autonomously, with 47 user-defined schema extensions, 0 compliance violations.

7 Limitations & Discussion

1. **The Specification Gap:** The system is only as safe as the completeness of constraints $\mathcal{C}$. We mitigate this through iterative human oversight.
2. **Convergence:** There is no guarantee that the LLM will find a valid plan within k_{max} iterations. The system defaults to "Fail-Secure".
3. **Meta-Validation Scope:** The Guard Resolver can only validate properties expressible in decidable logics (LIA, UF, Strings). Turing-complete properties remain undecidable.
4. **Specification Gaming at Meta-Level:** While the system prevents direct violations, a sophisticated adversary could potentially craft DSL extensions that exploit loopholes in the constraint set. We recommend formal verification of constraint sets as critical infrastructure.
5. **Scale Limits:** Current implementation tested on 47 auto-generated components. Behavior beyond 1000+ components remains unexplored.

8 Conclusion

CausaNova transforms Generative AI from a probabilistic assistant into a deterministic operator. By enforcing a mathematical firewall between intent and execution, and enabling recursive self-extension of its own logic, we demonstrate a practical approach to **Operational Alignment** in defined domains.

This is not a solution to AGI safety in the philosophical sense. Rather, it is an architectural pattern for deploying AI in high-stakes environments where formal guarantees can be provided. The system has been deployed in a production setting (German administrative compliance) and successfully generated thousands of artifacts without human oversight.

We present this work as both a technical contribution and an invitation to the research community: the problems we solved are real, but the challenges ahead are substantial. Future work must address formal verification of constraint sets, scaling beyond current limits, and integration with real-world legal and technical systems.