Logic-Guard-Layer

An Architecture Proposal for Deterministic Validation and Controlled Repair of Generative AI Outputs in Technical Domains

Martin Russmann
mrussmann@proton.me

December 2024
Revised March 2026

Abstract

Large language models produce fluent text, but fluency is not equivalent to correctness. This becomes a practical risk in technical domains, where outputs must satisfy value ranges, units, temporal constraints, referential integrity, and domain-specific terminology. This white paper proposes **Logic-Guard-Layer** (Logic-Guard-Layer), a neuro-symbolic middleware architecture that places a deterministic validation and repair layer between a generative model and an end application.

The proposal separates validation into two complementary stages. The first stage checks extracted claims against an ontology-informed constraint layer that captures structural and domain rules. The second stage checks eligible claims against external data sources through adapters with explicit source profiles. These profiles encode scope, source semantics, and operational status, enabling differentiated interpretation of missing or conflicting results. To support production use, the architecture uses a six-state validation status model: `MATCH`, `MISMATCH`, `ABSENCE`, `OUT_OF_SCOPE`, `LOOKUP_FAILURE`, and `UNKNOWN`.

A bounded repair loop is included as an optional component. It attempts correction only when the validation results provide actionable evidence and aborts when the process becomes unstable, exceeds an iteration limit, or degrades previously valid content beyond a configured threshold. The paper does not claim formal completeness or full description-logic reasoning. It presents an implementable architecture, a prototype scope, and an evaluation plan using real public APIs and structured public documents.

Keywords: Generative AI, LLM reliability, neuro-symbolic AI, ontology-informed validation, middleware architecture, factual verification, self-correction

1 Introduction

Large language models (LLMs) can summarize, explain, transform, and generate text with impressive flexibility. Their limitation is equally well known: they can produce outputs that are coherent in language but wrong in substance. In many consumer settings this is inconvenient. In technical settings it can be operationally harmful.

The core problem is architectural. An autoregressive model optimizes token likelihood, not truth, physical validity, or institutional accountability:

$$P(\mathbf{x}) = \prod_{t=1}^{n} P(x_t \mid x_1, \ldots, x_{t-1}) \tag{1}$$

This objective is powerful for language generation, but it does not by itself enforce that a reported measurement lies within a valid range, that a referenced entity exists, or that two dates are temporally consistent.

This proposal addresses the gap between linguistic generation and deterministic validation. Rather than expecting the model to guarantee correctness internally, Logic-Guard-Layer introduces an explicit external control layer that evaluates generated claims before they are accepted by a downstream application.

The proposal is motivated by technical domains that share three recurring characteristics: their outputs contain claims that can be checked against structural rules or authoritative sources, incorrect values or invented entities carry practical cost, and traceability matters more than stylistic fluency. Examples include operational reporting, technical documentation, public procurement extraction, environmental measurement summaries, and compliance-oriented information processing.

2 Context and Gap

Several existing approaches improve LLM reliability, but they leave an architectural gap that Logic-Guard-Layer is designed to fill.

Retrieval-augmented generation improves factual grounding by supplying documents at generation time [Lewis et al., 2020]. However, retrieval does not itself validate the generated answer. A model may still combine retrieved content incorrectly or introduce unsupported details. Structured output modes improve syntax and schema conformance at the format level, but a syntactically valid JSON object can still contain impossible values or false references. Guardrail frameworks are effective for policy filters, style restrictions, or unsafe content constraints, but they are usually not designed to validate domain-specific factual and structural correctness.

Recent hallucination research has improved fine-grained evaluation and external verification. FActScore decomposes text into atomic facts and evaluates support against knowledge sources [Min et al., 2023]. SelfCheckGPT uses response inconsistency across samples as a hallucination signal [Manakul et al., 2023]. FacTool integrates external tools for verification [Chern et al., 2023]. These approaches are highly relevant, but they primarily target detection and evaluation. They do not define a middleware architecture for deterministic validation and bounded repair in deployment settings.

Neuro-symbolic AI provides the broader intellectual context [Garcez et al., 2020, Hitzler and Sarker, 2022]. The specific problem here is narrower and more operational: how to insert a reproducible validation layer between a black-box LLM and an application that requires reliable outputs.

3 Design Goal and Positioning

Logic-Guard-Layer is proposed as middleware. It is not a new foundation model, not a training method, and not a claim of formal completeness. Its role is to mediate between probabilistic text generation and deterministic application requirements.

Concretely, the system is designed to identify claims in generated text that are suitable for deterministic checking, validate those claims against explicit domain constraints and selected external sources, distinguish semantic disagreement from missing coverage and technical failure, optionally trigger bounded repair when actionable evidence exists, and provide traceable results that can be inspected by an operator or downstream application. The system is therefore best understood as an *architecture proposal for post-generation validation and controlled repair*.

4 Architecture Proposal

4.1 System Boundary

Logic-Guard-Layer sits between an LLM and an end application. The LLM remains responsible for generation. The middleware is responsible for validation, aggregation, policy decisions, and

optional repair. The downstream application receives either an accepted output, a rejected output with traceable violations, or a warning-bearing output when evidence is incomplete.

The architecture is organized into six core components: a generation interface, a claim extraction module, a constraint validation layer, a source adapter layer, a result aggregation and policy engine, and an optional bounded repair loop. Each of these components is described in the subsections that follow.

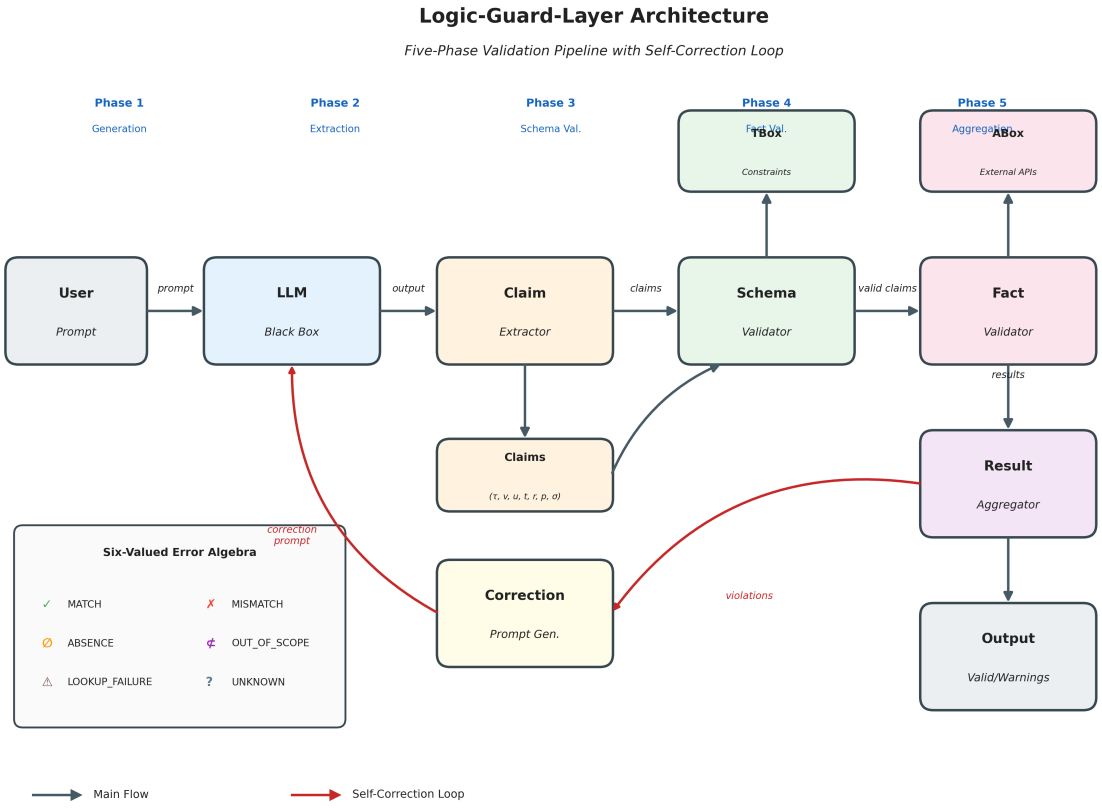

Figure 1: Proposed LOGIC-GUARD-LAYER architecture. The middleware extracts claims, validates them against constraints and selected data sources, aggregates results, and optionally initiates bounded repair.

4.2 TBox/ABox as Architectural Shorthand

The proposal uses the terms *TBox* and *ABox* as architectural shorthand for two kinds of checks. TBox-inspired checks enforce schema-level and rule-level constraints, while ABox-inspired checks perform instance-level verification against selected sources. This is a design distinction, not a claim that the full system implements OWL 2 DL reasoning end-to-end. In particular, external APIs are operational data sources, not pure assertional knowledge bases in the strict description-logic sense.

4.3 Claim Extraction

The first task is to transform generated text into units that can be checked. A claim is treated as a structured record:

$$c = (\tau, e, v, u, t, q, p) \tag{2}$$

where τ is the claim type, e is the referenced entity or subject, v is the value or proposition, u is the unit or representation, t is the time reference, q is a qualifier such as exact, approximate, or range, and p is provenance in the original output.

Claim extraction may combine three complementary mechanisms. LLM-based structured extraction handles flexible and context-sensitive parsing. Pattern-based extraction targets dates, units, values, and identifiers using deterministic rules. A parser-based fallback covers simple subject–predicate–object structures when neither of the first two approaches applies cleanly.

This stage is necessarily imperfect. Four operational error modes are relevant: false negatives (checkable claims that are missed), false positives (non-claims that are incorrectly extracted), semantic drift in extraction (claims whose meaning shifts during structuring), and granularity mismatch (claims extracted at a coarser or finer level than what the downstream validation can assess). Because the rest of the pipeline depends on extraction quality, extraction must be evaluated as a separate stage with its own metrics.

4.4 Constraint Layer

The first validation stage checks claims against an ontology-informed constraint layer. This layer is intentionally lightweight. It is not presented as a restricted subset of OWL 2 DL. It is a runtime-oriented constraint DSL informed by ontology concepts and domain modeling.

The constraint categories covered include type compatibility (whether the entity is of the right kind for the asserted property), unit compatibility (whether the stated unit is valid for the property), value-range plausibility (whether the value falls within physically or institutionally meaningful bounds), temporal consistency (whether dates and orderings are coherent), relation compatibility (whether the asserted relationship is meaningful for the given entity types), and required field presence for structured tasks.

A minimal abstract interface is:

$$\kappa_S(c, \mathcal{T}) \rightarrow (\texttt{pass/fail}, V_S) \tag{3}$$

where V_S is the set of schema-level violations for claim c.

A runtime-oriented DSL may use expressions such as:

```
entity_type(entity, type)
value_range(property, min, max, unit)
compatible_units(property, [unit1, unit2, ...])
temporal_order(event1, relation, event2)
requires(field_a, field_b)
```

The purpose of this layer is to reject structurally impossible or ill-formed claims before external lookup is attempted.

4.5 Source Adapter Layer and Source Profiles

The second validation stage checks eligible claims against selected external sources. Each source is wrapped by an adapter that standardizes access and returns both evidence and the metadata needed for interpretation.

A central improvement over the original version is that each source must have an explicit *source profile*:

$$\pi_d = (\text{scope}_d, \text{semantics}_d, \text{freshness}_d, \text{availability}_d, \text{authority}_d) \tag{4}$$

The source profile records five dimensions. **Scope** defines what the source is supposed to cover. **Semantics** specifies whether absence should be interpreted under a closed-world assumption (CWA-like), an open-world assumption (OWA-like), or a hybrid regime. **Freshness** captures the source's update behavior and temporal limits. **Availability** encodes the operational state relevant for retries and fallbacks. **Authority** classifies the source as normative, informative, or auxiliary.

This prevents a major conceptual error: a missing result cannot be interpreted correctly without knowing what the source is designed to contain and how absence should be read.

The fact-validation interface is therefore:

$$\kappa_F(c, d, \pi_d) \to (s, E_d) \tag{5}$$

where s is a validation status and E_d is the evidence returned by source d.

4.6 Six-State Validation Status Model

The original document described a six-valued error algebra. That label is too strong. What is actually needed for the architecture is a six-state validation status model.

Table 1: Validation status model used by Logic-Guard-Layer

Status	Operational meaning
`MATCH`	Claim is supported by the selected source within the configured tolerance.
`MISMATCH`	Claim conflicts with the selected source.
`ABSENCE`	The entity or asserted fact is absent in a source whose profile permits absence to be treated as negative evidence.
`OUT_OF_SCOPE`	The query is outside the declared scope of the source.
`LOOKUP_FAILURE`	The source could not be queried or interpreted reliably because of a technical failure.
`UNKNOWN`	The source does not provide enough information for a determinate judgment.

This distinction matters because different statuses imply different downstream actions. `MISMATCH` and `ABSENCE` are candidates for rejection. `LOOKUP_FAILURE` is a retry or fallback condition. `OUT_OF_SCOPE` and `UNKNOWN` propagate uncertainty and usually lead to warnings rather than hard rejection. The classification logic depends on the source profile, not just on the raw API response.

4.7 Aggregation and Decision Policy

The middleware must turn claim-level results into an output-level decision. A simple policy layer distinguishes between three categories: hard failures (structural impossibility, contradiction, or absence in a negative-evidence source), soft failures (unknown support or out-of-scope lookup), and operational failures (lookup failure, timeout, or rate limit).

The basic policy operates as follows. Outputs containing unresolved hard failures on mandatory claims are rejected. Outputs where all mandatory claims pass and warnings remain within tolerance are accepted. Soft failures are attached as warnings. When operational failures exceed a configured threshold, the system retries or defers to a fallback path.

This policy is configurable by application domain. A safety-critical application may reject more aggressively than a decision-support dashboard.

4.8 Bounded Repair Loop

When violations are actionable, the system may attempt repair. Repair is bounded and policy-controlled. It is not assumed to converge in the mathematical sense under real deployment conditions.

Let t_i be the current text and V_i the set of actionable violations. A repair step is:

$$t_{i+1} = \text{Repair}(t_i, V_i) \tag{6}$$

The repair prompt should include the original output or relevant excerpt, the identified violations, evidence or source-backed corrections when available, and an instruction to preserve unaffected content.

The loop terminates when no actionable violations remain, when the maximum number of iterations is reached, when a cycle is detected, when previously valid content changes beyond the allowed threshold, or when repeated lookup failures prevent meaningful progress. This is better described as a *bounded repair policy* than as a convergence theorem.

4.9 Non-Regression Signal

Repair may improve one claim while damaging another. To limit this, the architecture tracks a non-regression signal over the set of previously valid claims.

Let G_i^+ be the set of canonical representations of claims that were valid after iteration i. Then:

$$R_i = \frac{|G_i^+ \cap G_{i+1}^+|}{|G_i^+|} \tag{7}$$

and the corresponding drift score is

$$\Delta_i = 1 - R_i \tag{8}$$

This score is an operational heuristic, not a semantic proof. It depends on claim extraction and canonicalization being sufficiently stable. It is still useful as a practical abort signal: when Δ_i exceeds a configured threshold, the system can prefer the previous version.

5 Prototype Scope

The proposed prototype focuses on two validation tracks, chosen because they offer clearly defined claims, authoritative public sources, and distinct validation profiles.

5.1 Track A: Physical and Operational Measurements

This track validates claims about environmental or operational data using public APIs such as PEGELONLINE for water-level data [PEGELONLINE, 2024] and Bright Sky for weather data [Bright Sky, 2024]. The relevant constraints include valid entity identifiers, unit compatibility, plausible value ranges, timestamp validity, and tolerance handling for approximate values. Together these cover the most common modes of factual error when an LLM generates summaries of measurement data.

5.2 Track B: Structured Public Documents

This track validates claims extracted from public procurement notices using the TED Search API [TED, 2024]. The checks address deadline extraction, publication-date consistency, contract value extraction, identifier and code consistency, and basic document-field relations. These represent a different validation profile from Track A: the source is document-structured rather than measurement-structured, and many checks involve cross-field consistency rather than numeric plausibility.

5.3 Out of Scope

The prototype does not attempt full natural-language inference, irony or modality resolution, complex temporal logic beyond simple ordering, full multi-document contradiction analysis, or general-purpose truth adjudication across conflicting authorities. These are important future extensions, but excluding them makes the proposal technically credible by keeping the validation layer within the reach of deterministic methods.

6 Implementation Plan

A five-month implementation plan is realistic and maps directly onto the architectural components.

In the first month, the focus is on domain modeling and source profiling: defining claim types, domain constraints, source profiles, and adapter contracts for both prototype tracks. The second month covers claim extraction and the constraint engine, including structured extraction, pattern-based extraction, canonicalization, and the schema validation layer. The third month addresses

adapter implementation for the selected sources and the six-state status classification logic. The fourth month integrates the aggregation layer, policy engine, repair prompts, cycle detection, and non-regression control. The fifth month is reserved for evaluation, baseline comparison, latency measurement, failure-case documentation, and architectural refinement based on the experimental results.

7 Evaluation Plan

7.1 Evaluation Objective

The evaluation is designed to test whether the proposed middleware improves reliability relative to selected authoritative sources within the stated domain scope. It does not claim to establish universal truth.

7.2 Baselines

The prototype should be compared against four configurations of increasing complexity: LLM-only (no validation), constraint-only (only schema validation), source-only (only fact validation against external sources), and the full stack (extraction, constraints, source validation, aggregation, and optional repair). This decomposition allows the contribution of each architectural layer to be assessed independently.

7.3 Metrics

The evaluation should separate pipeline stages to isolate the contribution and failure modes of each component.

E0: Claim extraction quality. Precision, recall, and F1 of extracted claims against manually annotated samples.

E1: Validation performance. Detection quality for structural and factual violations, reported by type and status class.

E2: Repair performance. Success@k for valid outputs after at most k repair iterations, together with the non-regression score and drift.

E3: Operational performance. Latency at p50, p95, and p99, retry rate, and failure distribution across the six validation statuses.

7.4 Evaluation Questions

The prototype is successful if it provides evidence for four questions. First, does the full stack reduce invalid accepted claims compared with the baselines? Second, does the six-state status model reduce false rejection compared with naive missing-result handling? Third, does bounded repair improve output validity without unacceptable drift? Fourth, does the architecture remain operationally usable within the latency envelope of the target application?

7.5 Ground Truth Strategy

Ground truth must be defined relative to the selected sources and task scope. For extraction quality, the standard is manual annotation on a sampled subset. For source-backed claims, the configured authoritative source and tolerance rules define correctness. For ambiguous or out-of-scope cases, the evaluation records the validation status rather than forcing binary truth labels. This approach is methodologically cleaner than treating every missing result as false.

8 Risks and Limitations

The architecture has four principal limitations.

First, claim extraction remains a bottleneck. A validation layer can only assess claims that were extracted in a usable form. If extraction misses a claim or distorts its meaning, the downstream stages are blind to the error.

Second, the constraint library is never complete. The system is reliable only within the space of modeled rules and supported claim types. Constraints that have not been encoded cannot be enforced, no matter how obvious they might seem to a human reviewer.

Third, source authority is local and scoped. Even public APIs can be incomplete, delayed, unavailable, or semantically narrower than the generated claim. The source profile mechanism makes this explicit, but it does not eliminate the underlying limitation.

Fourth, repair remains probabilistic because the correcting model is still a language model. The architecture mitigates this through bounded repair and non-regression checks, but it does not eliminate the uncertainty entirely.

These limitations do not invalidate the proposal. They define the conditions under which it is credible.

9 Conclusion

Logic-Guard-Layer is best understood as a pragmatic architecture for adding deterministic control to generative AI in technical domains. Its contribution is not a claim of complete formal reasoning, but an implementable design pattern: extract claims, validate structure, validate selected facts, distinguish semantic disagreement from missing coverage and technical failure, and attempt repair only under bounded and inspectable conditions.

The proposal is intentionally modest in its claims and concrete in its engineering implications. That is a strength. It makes the architecture suitable for prototype implementation, comparative evaluation, and later extension into domain-specific production systems.

References

Chern, I.-C. et al. (2023). FacTool: Factuality Detection in Generative AI – A Tool Augmented Framework for Multi-Task and Multi-Domain Scenarios. *arXiv:2307.13528*.

Garcez, A. d'Avila et al. (2020). Neurosymbolic AI: The 3rd Wave. *arXiv:2012.05876*.

Hitzler, P. and Sarker, M. K. (2022). *Neuro-Symbolic AI: The State of the Art*. IOS Press.

Lewis, P. et al. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. *NeurIPS*, 33:9459–9474.

Manakul, P., Liusie, A., and Gales, M. J. F. (2023). SelfCheckGPT: Zero-Resource Black-Box Hallucination Detection for Generative Large Language Models. *EMNLP*, 9004–9017.

Min, S. et al. (2023). FActScore: Fine-grained Atomic Evaluation of Factual Precision in Long Form Text Generation. *EMNLP*, 12076–12100.

PEGELONLINE (2024). REST-API Documentation. https://pegelonline.wsv.de/webservice/dokuRestapi.

Bright Sky (2024). API Documentation. https://brightsky.dev/.

TED (2024). Search API Documentation. https://docs.ted.europa.eu/api/latest/search.html.