Logic-Guard-Layer

A Software Architecture for Deterministic Post-Generation Validation of Large Language Model Outputs

Martin Russmann
mrussmann@proton.me

January 2025

Executive Summary. Large Language Models (LLMs) produce linguistically plausible output by optimizing next-token prediction rather than factual correctness. This creates a structural reliability gap in operational settings where outputs must satisfy domain rules, numeric constraints, and external ground truth. Logic-Guard-Layer (LGL) is a software architecture designed to close that gap without modifying the underlying model. The architecture places a deterministic validation layer between model output and downstream use. It extracts structured claims from free-form text, validates them against formal constraints and authoritative data sources, classifies validation outcomes using a six-state decision model, and optionally triggers controlled repair while monitoring semantic drift. The central value of the architecture lies in its separation of concerns: probabilistic language generation remains isolated from rule enforcement, source interpretation, and audit logging. This document presents LGL as a reference architecture for enterprise and high-stakes LLM deployments, with emphasis on component boundaries, request flow, deployment patterns, operational safeguards, and acceptance criteria.

Keywords: LLM reliability, software architecture, neuro-symbolic validation, ontology-based checking, knowledge source integration, post-generation control, auditability

1. The Reliability Problem in LLM Systems

Large Language Models generate text by minimizing token-level prediction loss:

$$\mathcal{L}(\theta) = -\sum_{t=1}^{T} \log P(x_t \mid x_{<t}; \theta) \tag{1}$$

This objective produces strong fluency, but it does not guarantee correspondence with domain rules, physical limits, or current external data. The resulting failure mode is not random noise; it is a structural consequence of the optimization target. A model can produce text that is internally coherent, stylistically polished, and still operationally wrong.

Consider a concrete scenario. An LLM is tasked with summarising water level measurements along a federal waterway. It produces a report stating that Station Cologne recorded 3.45 m on a given date, that this represents a 12% increase over the previous week, and that the trend is consistent with seasonal norms. Every sentence reads well. But the actual measurement was 2.87 m, the percentage was computed from a hallucinated baseline, and the seasonal comparison references a period the model invented. None of this is syntactically detectable. The output conforms to every schema. It simply does not correspond to reality.

In low-stakes contexts—creative writing, brainstorming, casual summarisation—this may be acceptable. In operational environments, it is not. Maintenance reports, procurement summaries, regulatory statements, measurement descriptions, and decision-support outputs contain claims that can be checked against formal constraints or external sources. If those claims are wrong, downstream systems may act on false premises. The cost is not hypothetical: incorrect maintenance intervals can lead to equipment

failure, wrong procurement deadlines can invalidate tender submissions, and fabricated measurement values can corrupt compliance records.

The key architectural implication is straightforward: if generation is probabilistic and truth conditions are domain-specific, reliability cannot be delegated to the model alone. A separate validation layer is required—one that operates deterministically on the output after generation, rather than hoping the model gets it right during generation.

2. Why Current Mitigations Are Not Sufficient

Several widely used control patterns improve LLM behavior, but none of them provides deterministic post-generation validation on its own.

Retrieval-Augmented Generation (RAG) improves contextual grounding by exposing the model to relevant documents at inference time [7]. This is valuable, but it addresses input quality rather than output correctness. RAG does not guarantee that the model uses the retrieved material faithfully. A model can be given exact water level data and still produce a summary that rounds incorrectly, transposes timestamps, or invents a trend that the data does not support. Retrieval is necessary infrastructure; it is not a validation mechanism.

Schema-Constrained Output (JSON Schema, typed function calling) enforces syntactic structure. It ensures that a field named `water_level` exists and contains a number rather than a string. It does not ensure that the number is the correct one. A response like `{"water_level": 99.7, "unit": "m"}` is schema-valid but physically absurd for a river gauge in central Europe. Schema enforcement solves the formatting problem; it does not touch the truth problem.

Guardrail Frameworks such as NeMo Guardrails [8] can restrict topics, control tone, and filter unsafe categories. They are designed for behavioural boundaries: preventing the model from discussing prohibited topics, generating harmful content, or drifting off-task. They are not designed to verify that a claimed measurement value matches an authoritative source, or that a temporal ordering constraint holds between two dates in a procurement notice. Guardrails and domain validation are complementary concerns, not substitutes.

Human Review remains the implicit fallback in many deployments. A domain expert reads the output and judges whether it looks right. This works, but it is expensive, inconsistent across reviewers, difficult to scale, and provides weak traceability unless paired with structured evidence. Crucially, human review degrades under volume: the more outputs a reviewer must check, the more likely subtle errors pass unnoticed.

What remains missing is a control surface that interprets generated output as a set of checkable claims and subjects those claims to deterministic validation before release or downstream execution. LGL is designed to provide exactly this surface.

3. Architectural Principles

LGL is designed around five architectural principles that together define its role in an LLM deployment.

The first principle is **separation of concerns**. Language generation, rule evaluation, source access, and repair orchestration are isolated into distinct components with well-defined interfaces. The model knows nothing about the validation layer; the validation layer knows nothing about the model's internal weights or prompting strategy. This decoupling means that the model can be replaced, fine-tuned, or upgraded without touching the validation logic, and that rules can be updated without retraining anything.

The second principle is **deterministic control after probabilistic generation**. The model may remain stochastic—temperature settings, sampling strategies, and prompt variations are the model operator's concern. But the validation path that follows must be deterministic: the same claims,

checked against the same rules and the same source state, must produce the same verdict. This is not merely a software quality goal; it is a prerequisite for auditability.

The third principle is **source-aware epistemics**. Different knowledge sources have different completeness guarantees, and the architecture must respect this. A closed-world database that does not contain a record is making a definitive statement: the entity does not exist. An open-world knowledge graph that does not contain a record is making a much weaker statement: the entity may or may not exist. Treating both as "not found = false" is an engineering error with direct operational consequences, specifically a high rate of false alarms that erodes trust in the system.

The fourth principle is **composability**. New domains should be onboarded by adding ontologies, policies, and adapters rather than rewriting the core. The validation pipeline itself is domain-agnostic; domain knowledge enters through configuration.

The fifth principle is **auditability by design**. Every validation decision must be reconstructable from the extracted claims, the rules that were applied, the source responses that were received, and the final disposition. This is not a logging afterthought; it is a structural requirement that shapes component boundaries and data flow.

These principles frame LGL not as a replacement for the LLM, but as a supervisory architecture around it.

4. Reference Architecture

LGL is best understood as a middleware layer placed between LLM output and downstream consumers. Its purpose is not to generate content, but to transform generated text into an auditable decision process.

4.1 System Overview

The core architectural decision is the placement of a deterministic control plane around a non-deterministic generator. This control plane consists of four operational subsystems: an ingestion and extraction subsystem that converts free-form text into structured claims; a policy and schema subsystem that enforces domain rules; a source-resolution subsystem that checks claims against external data; and a decision and repair subsystem that aggregates outcomes, determines disposition, and optionally triggers bounded correction.

The orchestration API coordinates these subsystems and returns a validated disposition to the calling application. Importantly, no component in the pipeline modifies the original text silently. Every transformation—extraction, validation, repair—is explicit, logged, and reversible in the audit trail.

4.2 Component 1: Validation API Orchestrator

The Validation API is the entry point into the system. It receives model output together with contextual metadata such as the domain identifier, the adapter configuration, the validation mode (check-only or check-and-repair), and trace settings. It dispatches work to downstream components, manages timeouts and partial failures, aggregates their responses, and returns a structured validation result.

The orchestrator itself contains little domain logic. Its responsibilities are coordination, fan-out, timeout enforcement, error handling, and response assembly. This deliberate thinness makes the orchestrator stable across domain changes: when a new domain is onboarded, the orchestrator's code does not change—only the policy and adapter configurations that it reads at startup.

The API should expose a stable, versioned contract. Inputs include the raw text, a schema or domain identifier, optional source-selection overrides, and the desired validation depth. Outputs include the extracted claims, per-claim validation states with source evidence, the overall disposition (accept, reject, accept-with-warning, escalate, or repair), and a trace identifier linking to the full decision record.

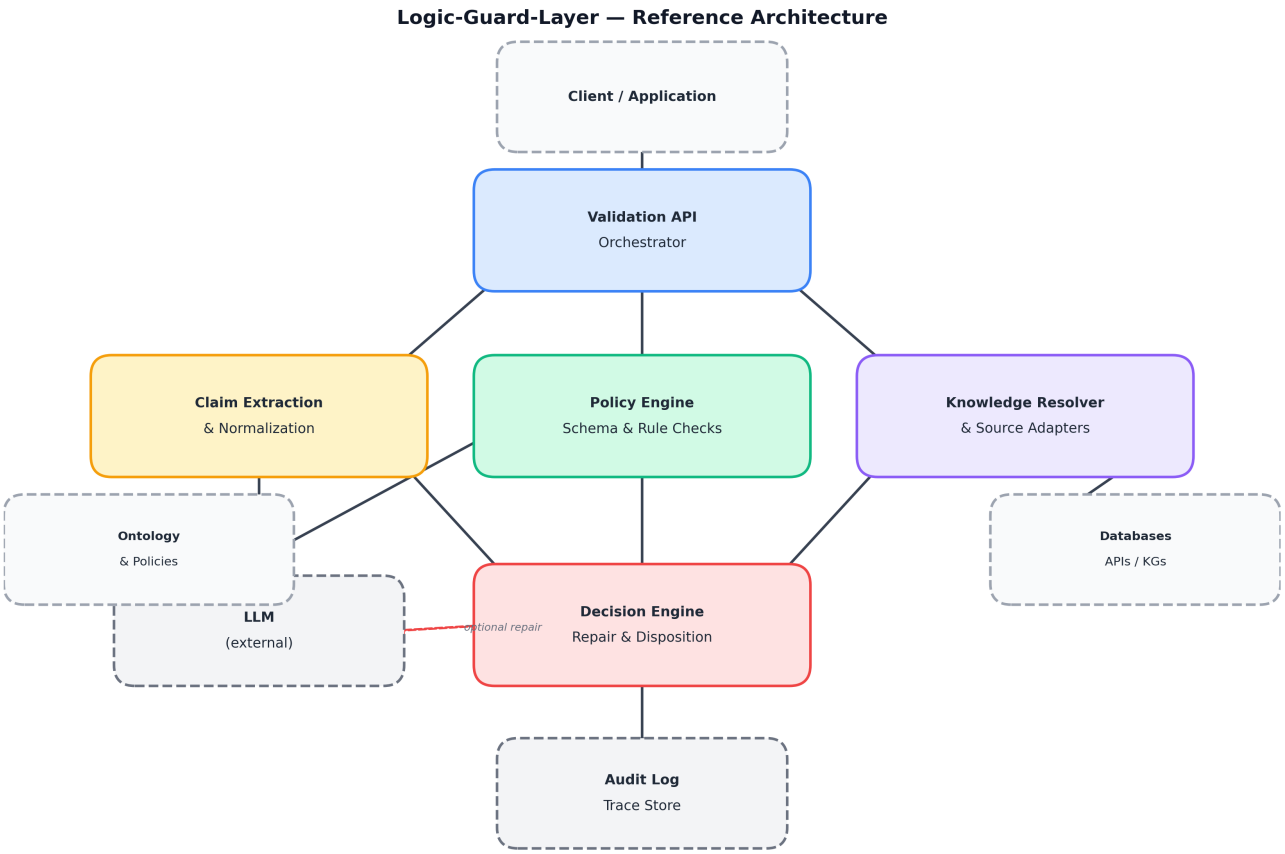

Figure 1: LGL as a supervisory middleware architecture. The LLM remains an external generation component; validation, source interpretation, and disposition are handled by dedicated control components.

4.3 Component 2: Claim Extraction and Normalization

The first substantive step is the conversion of free-form text into structured claims. In LGL, a claim is treated as a normalized record:

$$c = (s, p, o, u, \pi) \tag{2}$$

where s is the subject entity, p the predicate, o the object or value, u the unit of measurement, and π provenance metadata such as timestamp, source tag, and extraction context.

Extraction is the most ambiguous stage in the pipeline, because it must bridge the gap between natural language and formal structure. The architecture does not prescribe a single extraction method. Implementations may use LLM-based parsing with JSON-Schema constraints, rule-based NER followed by template matching, or hybrid approaches that combine both. The critical requirement is not the method but the output contract: all downstream components operate on structured claims, never on raw text. This contract is what makes the rest of the pipeline deterministic.

Claims are typed along four dimensions. *Numerical claims* assert a measured or computed value ("water level was 3.45 m"). *Temporal claims* assert dates, times, durations, or orderings ("submission deadline is March 15"). *Referential claims* assert entity identity or existence ("station KÖLN"). *Classificatory claims* assign categories or types ("CPV code 45233120"). Each type carries its own normalization logic: SI unit conversion for numerical claims, UTC normalization for temporal claims, identifier resolution for referential claims, and taxonomy lookup for classificatory claims.

The quality of extraction bounds the quality of everything downstream. A claim that is extracted incorrectly—a transposed digit, a misidentified entity, a lost temporal qualifier—will be validated against the wrong target. For this reason, extraction accuracy should be measured and reported separately from validation accuracy. Conflating the two makes diagnostic root-cause analysis impossible.

4.4 Component 3: Policy Engine

The Policy Engine performs deterministic checks against domain rules. This component is the rule-enforcement boundary of the system, and it should be understood as a composite subsystem rather than a monolithic reasoner.

In practice, three categories of rules are involved, and they differ in formalism, maintenance cadence, and authoring audience. *Ontology-level constraints* define entity types, predicate domains and ranges, and class hierarchies. They are expressed in OWL or a similar description logic and are maintained by knowledge engineers. *Shape and datatype constraints* define required fields, value types, cardinalities, and structural patterns. They correspond to SHACL shapes or JSON Schema and are often maintained by data architects. *Application rules* encode domain-specific business logic: numeric bounds (a river gauge in Germany will not read 99 m), temporal orderings (a submission deadline cannot precede a publication date), tolerance logic (a water level claim of 3.48 m is acceptable if the source says 3.45 m and the context uses approximate language), and cross-field invariants (maintenance interval must not exceed component lifespan).

Not every rule belongs in OWL reasoning. Putting a numeric range check into an ontological axiom is technically possible but architecturally wrong: it couples a fast, simple check to a slow, complex reasoning engine and makes the rule invisible to non-ontologists. The Policy Engine should therefore dispatch rules to the appropriate evaluator—ontology reasoner, shape validator, or application logic module—and aggregate results into a uniform violation report.

Each violation carries a type (schema or fact), a severity (error or warning), a human-readable message, and a reference to the constraint that was violated. This structured output feeds directly into the Decision Engine and, if repair is enabled, into the correction prompt generator.

4.5 Component 4: Knowledge Resolver and Source Adapters

The Knowledge Resolver handles external fact validation. It translates claims into source-appropriate queries, dispatches them to adapters, normalizes the responses, and returns source-level validation outcomes.

This subsystem isolates the rest of the architecture from the heterogeneity of external data. A PostgreSQL adapter speaks SQL. A SPARQL adapter constructs graph queries. A REST adapter calls HTTP endpoints with specific parameter conventions. An ERP adapter navigates proprietary APIs with authentication, pagination, and rate limiting. From the perspective of the validation pipeline, all of these look the same: a function that takes a claim and returns a result from the six-state algebra.

Each adapter declares four properties that the pipeline needs for correct interpretation. The *connector type* identifies the protocol. The *reliability weight* expresses how much the pipeline should trust this source relative to others—master data systems typically receive higher weight than secondary aggregators. The *timeout and retry policy* governs how long the pipeline waits and how many retries it attempts before classifying the outcome as a lookup failure. The *world assumption* declares whether the source operates under a closed-world assumption (absence is evidence of non-existence) or an open-world assumption (absence is lack of evidence). This last property is architecturally critical and is discussed further in Section 5.

Adapter queries execute concurrently via asynchronous I/O. In typical deployments, external source latency dominates all other processing stages. Parallel dispatch with bounded timeouts and circuit breakers is therefore not an optimization but an operational necessity.

4.6 Component 5: Decision Engine and Repair Coordinator

The Decision Engine is the final aggregation point. It receives rule outcomes from the Policy Engine and source outcomes from the Knowledge Resolver, merges them per claim, applies severity and precedence logic, and produces an overall disposition.

The disposition is not binary. The engine distinguishes five outcomes: *accept* (all claims passed or produced only informational results), *reject* (one or more hard errors with no repair enabled), *accept-with-warning* (no hard errors but unresolved uncertainty states that should be visible to the consumer), *escalate* (the system cannot make a confident determination and routes to human review), and *repair* (hard errors detected, repair policy is active, and the system will attempt bounded correction).

When repair is enabled, the Decision Engine generates a structured correction prompt that identifies only the violated claims, provides the expected values or constraint descriptions from the validation evidence, and explicitly instructs the model to preserve all non-violated content. The repaired output then re-enters the pipeline at the extraction stage and is validated again. This loop is bounded by the safeguards described in Section 7.

Repair is optional by design. In many operational environments, detection and flagging is the correct response; automated correction introduces its own risks. The architecture supports both modes and lets deployment policy decide.

5. Validation State Model

A binary true/false model is not adequate for heterogeneous source validation. When a system queries five different sources about a single claim and receives two confirmations, one contradiction, one timeout, and one "not applicable," collapsing this into a boolean loses exactly the information that matters for operational decisions. LGL therefore uses a six-state validation model.

$$\mathcal{R} = \{\textsc{Match}, \textsc{Mismatch}, \textsc{Absence}, \textsc{OutOfScope}, \textsc{LookupFailure}, \textsc{Unknown}\} \tag{3}$$

Table 1: Six-state validation model and operational interpretation.

State	Severity	Operational meaning
MATCH	Informational	The claim is confirmed by an applicable source or rule.
MISMATCH	Error	The claim contradicts an applicable source or a deterministic rule.
ABSENCE	Error	The referenced entity or value is absent in a source declared authoritative under closed-world semantics.
OUTOFSCOPE	Warning	The source cannot answer because the query lies outside the source's declared coverage.
LOOKUPFAILURE	Warning	The source could not be queried successfully because of network, timeout, or infrastructure failure.
UNKNOWN	Warning	Available evidence is insufficient to confirm or refute the claim, typically under open-world semantics.

The practical importance of this model becomes clear through examples. Suppose a claim references a water level measurement from 1995. The PEGELONLINE API, which provides federal waterway data, has records starting from approximately 2006 for most stations. A naive binary system would mark the claim as "not found" and classify it as false. This is wrong—the source simply cannot answer the question. The correct classification is OUTOFSCOPE: the query lies outside the temporal coverage of the source, and no conclusion about the claim's truth value can be drawn. Similarly, a query for a station identifier that does not appear in the authoritative station registry is genuinely ABSENT—the closed-world database is making a definitive statement. And a query that times out after 30 seconds is a LOOKUPFAILURE—an infrastructure event that says nothing about the claim itself.

This distinction is one of the most important control features in the architecture. It prevents a common systems error: treating every `NOT_FOUND` outcome as if it were a hard contradiction.

5.1 Closed-World and Open-World Interpretation

The architecture assigns semantics to missing data through source policy rather than through ad hoc downstream interpretation:

$$\text{interpret}(\texttt{NOT\_FOUND}, a) = \begin{cases} \textsc{Absence} & \text{if } a \text{ is CWA} \\ \textsc{Unknown} & \text{if } a \text{ is OWA} \end{cases} \tag{4}$$

where a denotes the adapter that produced the response. A PostgreSQL table of registered equipment operates under CWA: if motor M1 is not in the table, motor M1 does not exist in the organisation's asset registry. A SPARQL knowledge graph compiled from heterogeneous public sources operates under OWA: if motor M1 is not in the graph, the graph simply lacks the record.

This policy-based interpretation allows the same low-level HTTP 404 or empty result set to carry different epistemological weight depending on the source's declared guarantees. The interpretation is configured at the adapter level, not inferred at runtime, which makes it auditable and predictable.

5.2 Source Aggregation

When multiple sources are available for the same claim, LGL aggregates source-level outcomes using a weighted scoring mechanism:

$$\text{score}[r] = \sum_{a \in A} w_a \cdot \mathbf{1}[\text{result}(a, c) = r] \tag{5}$$

where w_a is the configured reliability weight of adapter a and $r \in \mathcal{R}$.

Pure arithmetic aggregation is not sufficient for operational safety. The Decision Engine therefore applies a precedence policy on top of the weighted scores. A single MISMATCH from a high-trust, applicable source constitutes a blocking signal regardless of how many lower-weight sources return MATCH. A single ABSENCE from an authoritative closed-world registry blocks the claim even if open-world sources return UNKNOWN. Conversely, a collection of UNKNOWN, OUTOFSCOPE, and LOOKUPFAILURE results from all queried sources does not constitute a hard error—it constitutes unresolved uncertainty that may warrant escalation to a human reviewer but should not trigger automated rejection.

This layered approach—weighted scoring for soft ranking, precedence rules for hard boundaries—makes the aggregation logic both configurable and defensible under audit.

6. Request Lifecycle

A typical LGL transaction proceeds through a defined sequence of stages, each producing an explicit, logged output that feeds into the next.

The lifecycle begins with **input reception**: the client submits model output together with a domain identifier and optional configuration overrides. The orchestrator parses the request, validates the configuration, and initialises a trace record.

Claim extraction follows. The raw text is transformed into a set of normalised claims, each carrying subject, predicate, object, unit, and provenance metadata. The extraction component may invoke an LLM or a rule-based parser; the output is always a structured claim set.

The claims then enter **parallel validation**. The Policy Engine evaluates structural, numeric, temporal, and ontological rules against each claim. Simultaneously, the Knowledge Resolver dispatches applicable claims to configured source adapters. Both subsystems return typed violation or confirmation records.

State classification maps rule outcomes and source outcomes into the six-state model described in Section 5. Each claim now carries a composite validation state reflecting both schema compliance and factual verification.

The **disposition stage** applies severity logic and precedence rules to produce a system-level outcome: accept, reject, accept-with-warning, escalate, or repair. If repair is triggered, the system generates a constrained correction prompt, sends it to the model, and re-validates the result—repeating until convergence or abort.

Finally, **trace persistence** writes the complete decision record to the audit store: extracted claims, rule evaluations, source responses, aggregation logic, disposition, and (if applicable) the repair history. This record is sufficient to reconstruct the decision from scratch, which is the auditability guarantee that Section 11 depends on.

The architectural advantage of this flow is that every transition is explicit and inspectable. The system does not move directly from generated text to business action. There is always an intermediate representation (claims), a validation verdict (states), and a documented disposition (accept/reject/escalate) between generation and use.

7. Controlled Repair and Drift Safeguards

Repair is treated as a bounded orchestration feature, not as autonomous reasoning. The model receives a structured correction prompt that identifies only the violated claims, provides the expected values or constraint descriptions from validation evidence, and instructs preservation of unaffected content. The repaired output is then re-validated under the same rules.

This sounds straightforward, but iterative correction of stochastic outputs introduces failure modes that the architecture must actively prevent.

7.1 Cycle Detection

The simplest pathology is oscillation: the model fixes error A but introduces error B, then fixes B but reintroduces A, and the loop never terminates. LGL prevents this by hashing the canonical claim set at each iteration and maintaining a history of observed states. If a hash recurs, the system has entered a cycle. It terminates immediately and returns the best intermediate result—the iteration with the fewest hard violations—rather than continuing indefinitely.

7.2 Maximum Iteration Bound

Even without detectable cycles, a loop may fail to converge within a useful time budget. Every repair loop is therefore bounded by a hard maximum iteration count $n_{\max}$, typically set between 3 and 5. This converts potentially unbounded stochastic behaviour into bounded operational behaviour. When the maximum is reached without convergence, the system returns the best available state with a clear non-convergence flag.

7.3 Semantic Drift Threshold

The most insidious failure mode is semantic drift: the model fixes the targeted violation but silently modifies claims that had already passed validation. A correction that fixes an operating-hours value but quietly changes a weight specification from 450 kg to 380 kg has introduced a new error while resolving an old one. In safety-critical domains, this is worse than leaving the original error in place, because the drift is invisible without re-validation.

LGL monitors drift using a formal metric:

$$\Delta_i = 1 - \frac{|\mathcal{S}_i^+ \cap \mathcal{S}_{i+1}^+|}{|\mathcal{S}_i^+|} \tag{6}$$

where $\mathcal{S}_i^+$ denotes the set of claims that were not in hard violation at iteration i. A drift of zero means perfect non-regression: every claim that was correct before remains correct after. A drift of one means total regression: every previously correct claim was modified.

The architecture enforces a configurable drift threshold $\delta_{\max}$, typically 5%. When $\Delta_i > \delta_{\max}$, the repair loop aborts and reverts to the previous state. The design philosophy is explicit: in high-stakes environments, preserving a partially correct state is preferable to introducing collateral damage in pursuit of full correction.

8. Implementation Blueprint

LGL can be implemented as a stateless service with asynchronous I/O and policy-driven plug-ins. The following blueprint reflects the technology choices that have proven effective in prototype development.

8.1 Interface Contract

The API exposes a single validation endpoint (`POST /validate`) with a stable, versioned contract. The request carries the raw text, a domain or schema identifier, an optional adapter-set override, a validation

Table 2: Illustrative implementation blueprint.

Layer	**Suggested technology**	**Purpose**
API layer	FastAPI or equivalent	Request orchestration and response contracts
Claim extraction	LLM + parser hybrid	Structured claim generation
Rule engine	OWL / SHACL / app logic	Deterministic constraint checking
Source adapters	Async Python connectors	External fact validation
Caching	Redis	Hot ontology, policy, and query caching
Trace store	SQL / document store	Auditability and replay
Deployment	Containers / Kubernetes	Horizontal scaling and isolation

mode flag (`check` or `check_and_repair`), and a trace flag. The response returns the normalised claims, per-claim validation states with source evidence, the overall disposition, any repaired output if correction was triggered, and a trace identifier that links to the complete decision record in the audit store.

This contract is deliberately minimal. It does not expose internal component boundaries to the caller, which allows the implementation to evolve—swapping a rule engine, adding an adapter, restructuring the extraction pipeline—without breaking the API surface.

8.2 Concurrency and Resilience

The orchestrator runs source adapters concurrently via asynchronous dispatch. In typical deployments, external source latency dominates all other processing stages by an order of magnitude: ontology reasoning completes in under 50 ms, while an external API call may take 500 ms to 2 s. Parallel dispatch with bounded timeouts is therefore not an optimisation but a necessity for meeting latency targets.

Each adapter connection is wrapped in a circuit breaker. After a configurable number of consecutive failures, the circuit opens and the adapter is temporarily excluded from the query set. Its claims are classified as LookupFailure rather than blocking the pipeline. This graceful degradation model ensures that a single unstable source does not compromise the validation of claims that depend on other, healthy sources.

Retry logic uses exponential backoff with jitter to avoid thundering-herd effects on recovering sources. The retry budget is bounded: no adapter may consume more than its declared timeout, regardless of how many retries are attempted within that window.

8.3 Deployment Topology

The architecture is designed for containerised deployment. Stateless API pods sit behind a load balancer, sharing access to a Redis cache that holds hot ontologies, compiled policies, and frequently queried source results. Adapter pools are isolated per source to prevent a slow or failing source from consuming thread resources needed by healthy sources. The trace store is a separate persistence layer, optionally fed asynchronously to avoid adding write latency to the critical validation path.

This topology supports horizontal scaling without entangling core validation logic with infrastructure concerns. Adding capacity means adding pods; adding a domain means adding policy and adapter configuration; adding a source means adding an adapter deployment. None of these changes require modification of the core validation pipeline.

9. Domain Onboarding and Extensibility

A new domain should be introduced through configuration and plug-in extension rather than core rewrites. In practical terms, onboarding a domain requires four assets: a domain vocabulary and policy set that defines entity types, predicates, and their relationships; structural and semantic constraints

expressed as ontology axioms, shape definitions, or application rules; source adapters configured for the domain's authoritative data systems; and claim extraction patterns appropriate to the domain's characteristic text structures.

This separation keeps the architecture general while preserving domain specificity where it belongs—in configuration artifacts rather than in code. The maintenance model is also clear: domain experts maintain policies and ontologies, integration engineers maintain adapters, and the core development team maintains the pipeline. These three concerns evolve at different rates and should not be coupled.

Practical domain examples illustrate the range of applicability. *Measurement and monitoring* domains (water levels, weather data, sensor readings) produce numerical and temporal claims validated against time-series APIs. *Public procurement* domains (tender notices, contract awards) produce referential and temporal claims validated against structured document registries like TED. *Financial reporting* domains produce numerical claims with strict temporal versioning requirements validated against audited datasets. *Regulated document generation* domains (compliance summaries, controlled medical reports) produce classificatory and referential claims validated against regulatory ontologies. In each case, the validation pipeline is the same; only the policy surface changes.

10. Validation Strategy and Operational KPIs

This document describes a reference architecture, not a completed benchmark report. Validation is therefore framed in terms of operational acceptance criteria rather than claimed empirical results.

10.1 Functional Correctness

A credible pilot must assess four dimensions independently. *Claim extraction accuracy* measures whether the extraction component correctly identifies entities, predicates, values, and units from the input text; errors here propagate to every downstream stage and must be measured in isolation. *Rule violation detection* measures whether the Policy Engine correctly identifies structural, numeric, temporal, and ontological constraint violations. *Source contradiction detection* measures whether the Knowledge Resolver correctly identifies mismatches between claimed values and authoritative source data. *Uncertainty classification* measures whether the six-state model correctly distinguishes Absence from OutOfScope from Unknown from LookupFailure—the distinctions that prevent false alarms. Each dimension should be reported separately; an aggregate accuracy number that conflates extraction errors with validation errors is diagnostically useless.

10.2 Operational Performance

Representative service-level targets include p_{95} end-to-end latency below 3 seconds for a typical validation request, bounded timeout behaviour under source instability (no request should hang indefinitely regardless of adapter state), and graceful degradation under partial source failure (the system should still produce a useful validation result when one of several sources is unavailable, classifying the affected claims as LookupFailure rather than blocking the entire response).

10.3 Repair Quality

If repair is enabled, the pilot should measure Success@k—the fraction of cases where all hard violations are resolved within k iterations—with a target above 90% at $k = 3$. Drift should remain below 5% per iteration. No uncontrolled loops should occur: every repair invocation must terminate within $n_{\max}$ iterations, and cycle detection must prevent infinite oscillation.

10.4 Audit and Governance

Finally, the architecture should be judged on trace completeness (every decision is reconstructable), reproducibility (the same inputs and source state produce the same outputs), policy transparency (every rule is inspectable and attributable), and ease of post-incident review (an auditor can follow the decision trail from input to disposition without requiring access to model internals).

A rigorous benchmark must separate claim extraction failure from downstream validation failure. Without that separation, the system cannot be diagnosed correctly, and improvement efforts will be misdirected.

11. Governance, Control, and Auditability

One of the most practical advantages of LGL is the conversion of opaque model output into a structured decision trail. This matters in environments where reliability is not only a technical requirement but also a regulatory or contractual one.

The architecture supports governance through four mechanisms. First, *explicit policy boundaries*: rules and source semantics are configured declaratively rather than hidden inside prompt engineering. A regulator or auditor can inspect the policy set and understand what the system checks without reading model weights or prompt templates. Second, *evidence-linked decisions*: every blocking or warning state is tied to a specific rule violation or source response with a traceable identifier. The question "why was this output rejected?" always has a concrete, inspectable answer. Third, *deterministic post-generation checks*: the decision path from extracted claims to final disposition is fully deterministic and reproducible. Running the same claims through the same policy and source state produces the same verdict, which is a prerequisite for regulatory certification in many industries. Fourth, *escalation support*: uncertainty states such as UNKNOWN and OUTOFSCOPE can route to human review rather than being collapsed into false certainty. The system explicitly represents the limits of its own knowledge, which is a form of institutional honesty that manual review processes often lack.

Together, these mechanisms make LGL suitable for organisations that require documented, auditable control over automated language output—whether for internal quality assurance, external regulatory compliance, or contractual liability management.

12. Risks, Assumptions, and Boundaries

LGL reduces a specific class of reliability failures, but it does not eliminate all sources of error. Clarity about the architecture's boundaries is as important as clarity about its capabilities.

Claim extraction remains the primary bottleneck. If the extraction stage misses a claim, misidentifies an entity, or distorts a numeric value, downstream validation operates on a corrupted representation. The pipeline can only validate what it can see. This is why extraction accuracy must be measured separately and why investment in extraction quality yields disproportionate system-wide returns.

Formalised coverage is necessarily incomplete. The system validates claims against encoded rules and configured sources. Domain knowledge that has not been formalised—implicit expertise, contextual judgment, common-sense reasoning—remains outside the control boundary. The architecture does not claim to catch all errors; it claims to catch the errors that fall within its formalised scope, and to do so deterministically.

Ontology and policy maintenance is a real cost. Creating a domain ontology requires subject-matter expertise. Maintaining it as the domain evolves requires ongoing effort. This cost is justified in high-stakes domains where the alternative—unchecked LLM output—carries higher risk. But it should be budgeted explicitly rather than treated as a one-time setup task.

Source quality constrains validation certainty. An adapter can only be as reliable as the

source it connects to. If a source contains stale data, the validation may confirm a claim that is no longer true. If a source revises historical records, previously validated claims may retroactively become incorrect. The architecture mitigates this through source weighting, multi-source consensus, and explicit uncertainty classification, but it cannot compensate for fundamentally unreliable ground truth.

Repair is bounded, not guaranteed. The repair loop improves usability when it succeeds, but it operates on a stochastic system (the LLM) and cannot guarantee convergence. The safeguards—cycle detection, iteration limits, drift thresholds—convert this uncertainty into bounded, predictable behaviour. But they do not convert it into certainty.

These limitations are not defects of the architecture. They define its operational scope. A system that claims to solve all reliability problems is not credible; a system that clearly delineates what it does and does not guarantee is deployable.

13. Conclusion

Logic-Guard-Layer is best understood as a software control architecture for reliable use of LLMs in operational settings. Its central design move is simple but consequential: generated text is not treated as a finished artifact, but as an intermediate object that must pass through structured extraction, deterministic validation, source-aware interpretation, and explicit disposition before release.

The architectural value lies in controlled separation. The LLM remains responsible for language generation. LGL assumes responsibility for rule enforcement, source integration, uncertainty classification, repair orchestration, and traceability. This separation allows probabilistic generation to be used inside systems that still require deterministic control surfaces—which describes most enterprise environments where LLMs are being deployed today.

The six-state validation model addresses a gap that binary fact-checking cannot fill: the need to distinguish genuine contradiction from absence, out-of-scope queries, infrastructure failures, and epistemic incompleteness. By encoding world assumptions at the source level and propagating them through the pipeline, the architecture avoids the false-positive inflation that erodes trust in automated validation systems.

The controlled repair mechanism extends the architecture from detection to correction, while the drift metric and cycle detection ensure that correction does not introduce more problems than it solves. The explicit audit trail converts every validation decision into a reconstructable, inspectable record.

As a reference design, LGL is not a claim that LLMs can be made intrinsically truthful. It is a claim that their outputs can be placed inside a stricter software architecture—one where factual risk is reduced, uncertainty is classified rather than hidden, and operational decisions are auditable. In domains where being right matters more than sounding right, that architecture is not optional. It is infrastructure.

References

[1] A. d'Avila Garcez, L. C. Lamb. Neurosymbolic AI: The 3rd wave. *arXiv:2012.05876*, 2020.

[2] G. Marcus. The next decade in AI: Four steps towards robust artificial intelligence. *arXiv:2002.06177*, 2020.

[3] P. Hitzler, M. K. Sarker. *Neuro-Symbolic Artificial Intelligence: The State of the Art*. IOS Press, 2022.

[4] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, P. Fung. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38, 2023.

[5] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin, T. Liu. A survey on hallucination in large language models. *arXiv:2311.05232*, 2023.

[6] S. Pan, L. Luo, Y. Wang, C. Chen, J. Wang, X. Wu. Unifying large language models and knowledge graphs: A roadmap. *IEEE TKDE*, 36(7):3580–3599, 2024.

[7] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, D. Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. *NeurIPS*, 2020.

[8] T. Rebedea, R. Dinu, M. Sreedhar, C. Parisien, J. Cohen. NeMo Guardrails: A toolkit for controllable and safe LLM applications. *EMNLP (Demos)*, 2023.

[9] A. Madaan, N. Tandon, P. Gupta, et al. Self-refine: Iterative refinement with self-feedback. *NeurIPS*, 2023.

[10] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, Z. Wang. HermiT: An OWL 2 reasoner. *J. Automated Reasoning*, 53(3):245–269, 2014.