LLM & SOA

Towards Sustainable ROI from Large-Scale LLM Systems

Pierre Bonnet, founder of the community www.engage-meta.com

pierre.bonnet@hlfl-consulting.com

September 03, 2025

Because LLMs exhibit non-deterministic behavior, their execution must be constrained by pre- and post-conditions, in line with the classical Service-Oriented Architecture (SOA) pattern.

Context

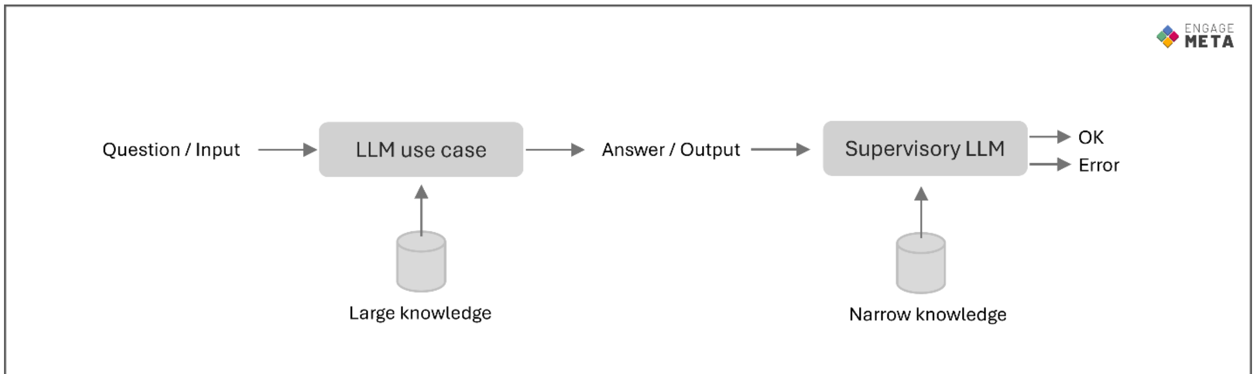
The ability of LLMs to generate code gives them the capacity to design and execute processes on the fly. This development pipeline — user prompt → code generation → agentic execution — opens the door to building highly sophisticated, “super-intelligent” information systems. By dynamically producing code, such systems can access databases, consume APIs, perform mathematical computations, and interact with workflows, all while adapting to changing business and organizational contexts.

Yet the non-deterministic nature of LLMs makes unit and integration testing virtually impossible. This limitation blocks large-scale adoption and undermines any prospect of return on investment. Put simply, no IT professional will risk deploying applications or automations that can randomly produce bugs or hallucinations.

The remainder of this paper explores solutions to mitigate — and even eliminate — these risks.

Approach 1 – Supervisory LLM

One approach is to deploy an LLM (a) dedicated to monitoring the outputs of another LLM (b) used within applications. For this setup to yield meaningful results, two conditions must be met. First, there must be complete semantic isolation between (a) and (b). In other words, no knowledge should be shared: each model must be trained on distinct datasets and knowledge domains. Second, LLM (a) must be constrained to operate within a sufficiently narrow and formalized knowledge space, thereby reducing the likelihood of hallucinations when subject to oversight. Unfortunately, these two conditions cannot realistically be guaranteed.



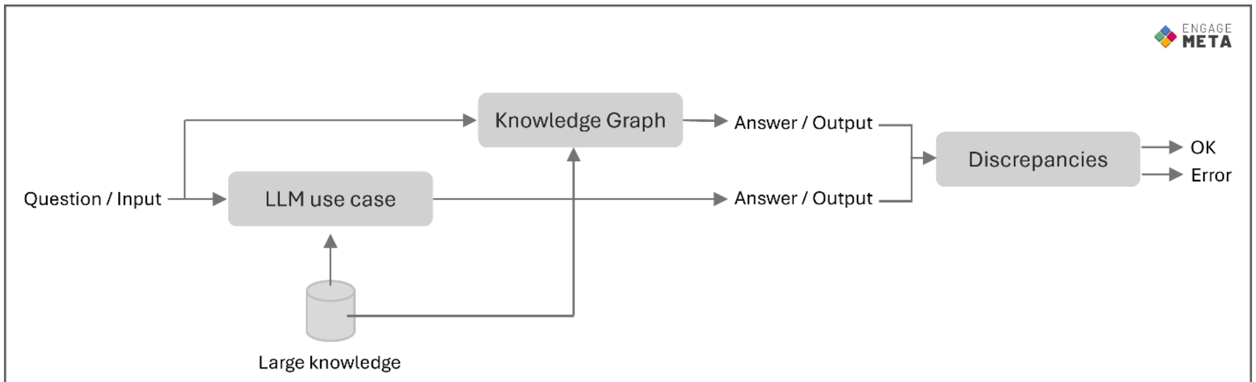
Approach 1 – Supervisory LLM

In practice, using an LLM to supervise the behavior of an application system can only serve at a high-level, holistic layer — detecting unusual execution patterns or supporting broad regulatory monitoring. As such, this solution cannot be relied upon to control hallucinations within LLM-driven applications and automations.

Approach 2 – Knowledge Graph

Another approach is to represent the semantic scope of each LLM use case as a formal knowledge graph. In this setup, the documentation used to train an application-specific LLM is also ingested into a knowledge-graph database with the appropriate semantic granularity (see our website documentation on prompt-driven graph generation).

With this configuration, a user query — whether submitted through a chat interface or triggered via automation — is sent simultaneously to the LLM (a) and translated into a deterministic query against the knowledge graph (b). While the LLM's response (a) may contain hallucinations, the graph's response (b) is deterministic, since it is confined to the semantic scope of nodes and their relationships. Comparing (a) and (b) makes it possible to detect discrepancies and infer potential hallucinations.



Approach 2 – Knowledge Graph

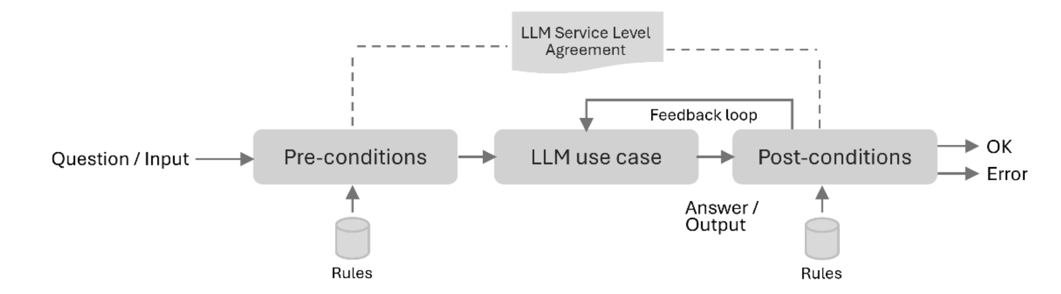
This mechanism can be further reinforced by adding queries to operational databases. In practice, however, this RAG-like approach remains challenging to implement: it requires a supervisory intelligence capable of identifying hallucinations by contrasting LLM outputs with more deterministic data sources. At best, it provides an analytical report on the reliability of the LLM's answers — but it cannot guarantee hallucination-free outputs.

Approach 3 – Service Oriented Architecture (SOA)

The third approach builds on a service-oriented architecture (SOA) pattern, in which each LLM use case is encapsulated within a service contract. This contract formalizes the usual parameters that govern LLM behavior (role, context, objectives, examples, etc.) while adding explicit pre- and post-conditions designed to control hallucinations.

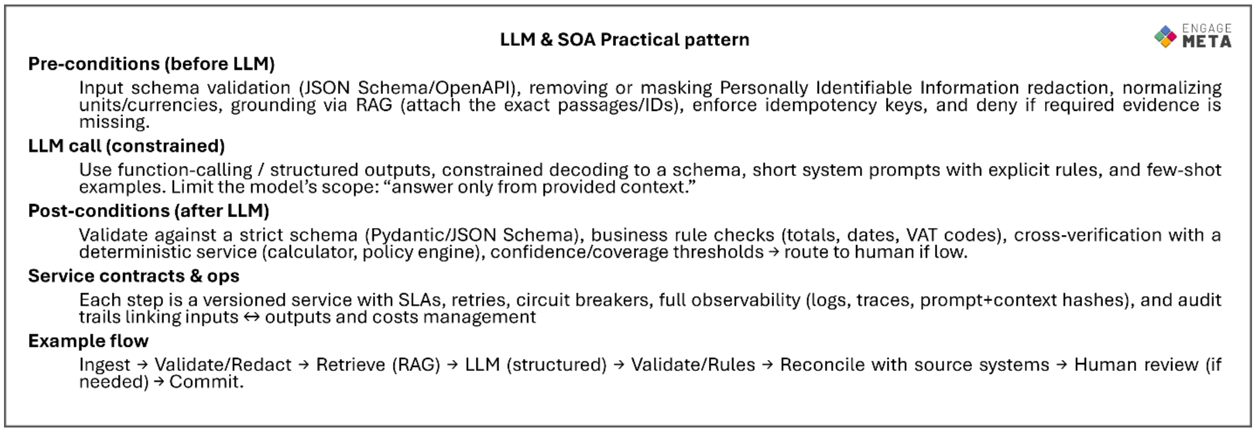
These conditions are enforced through deterministic rules that do not rely on LLMs. This pattern delivers two key benefits: reusability (1) and security (2):

1. Reusability. Encapsulating each LLM use case within a service backed by an explicit contract encourages reuse across multiple applications and automations. It effectively creates a logical architecture of LLM sequences that interact not only with each other but also with traditional, non-LLM services. Such an architecture is critical for keeping control over automations, as many cases documented online are too complex to be industrialized in real enterprises. For instance, it is common to see N8N automations with more than ten steps that cannot be maintained at scale. These must be restructured into elementary logical components — services with contracts that clearly specify invocation conditions (pre-conditions) and output expectations (post-conditions).
2. Security. A core function of the service contract is to guarantee data security both at invocation and at result delivery. For example, anonymization rules naturally belong to pre-conditions, as do checks on user authorizations. On the output side, post-conditions enforce data security by validating quality. This could mean ensuring that no non-professional content appears in text generated by an LLM, or that numerical outputs respect defined threshold values.



Approach 3 – Service Oriented Architecture (part 01)

The figure below illustrates a possible pattern:



LLM & SOA Practical pattern

Pre-conditions (before LLM)

Input schema validation (JSON Schema/OpenAPI), removing or masking Personally Identifiable Information redaction, normalizing units/currencies, grounding via RAG (attach the exact passages/IDs), enforce idempotency keys, and deny if required evidence is missing.

LLM call (constrained)

Use function-calling / structured outputs, constrained decoding to a schema, short system prompts with explicit rules, and few-shot examples. Limit the model's scope: "answer only from provided context."

Post-conditions (after LLM)

Validate against a strict schema (Pydantic/JSON Schema), business rule checks (totals, dates, VAT codes), cross-verification with a deterministic service (calculator, policy engine), confidence/coverage thresholds → route to human if low.

Service contracts & ops

Each step is a versioned service with SLAs, retries, circuit breakers, full observability (logs, traces, prompt+context hashes), and audit trails linking inputs ↔ outputs and costs management

Example flow

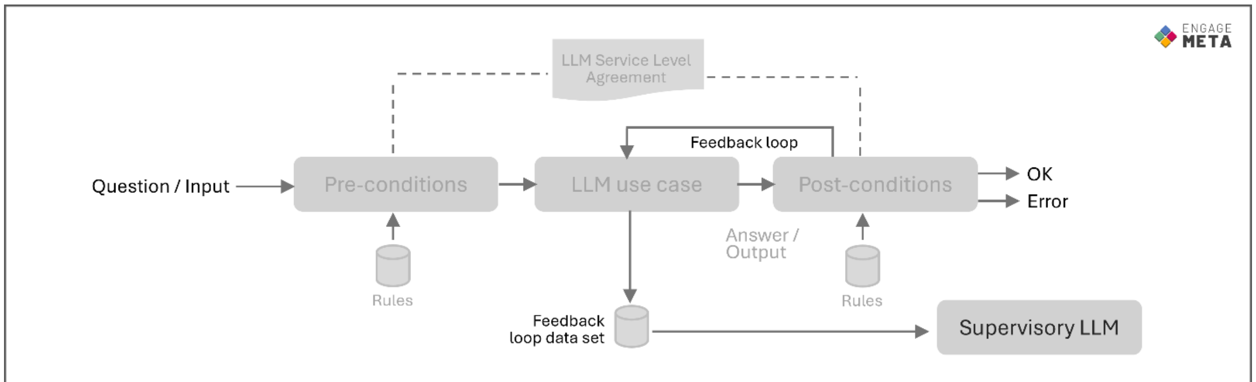
Ingest → Validate/Redact → Retrieve (RAG) → LLM (structured) → Validate/Rules → Reconcile with source systems → Human review (if needed) → Commit.

As noted earlier, the rules embedded in a service contract must not rely on the LLM itself; this ensures a deterministic, secure, and predictable behavior. This requirement does not imply that only rigid programming languages can be used. Drawing on years of SOA implementation, we know how to design data-driven rules (via parameterization) as well as deploy rule engines. It is also feasible to leverage symbolic AI derived from knowledge graphs, combined with heuristics, as a complementary technology for implementing the pre- and post-conditions of LLM services.

Learning System

Although an LLM service contract relies on deterministic pre- and post-conditions, this pattern can still be coupled with a supervisory LLM to assess the effectiveness of those contracts. For example, when a post-condition triggers a blocking error, two outcomes are possible: the process may stop with an error returned to the calling system (e.g., an error message displayed to the user), or the system may re-execute the LLM service after dynamically adjusting certain execution-context data to guide the LLM in correcting its output.

This feedback loop generates a record in a dataset that the supervisory LLM can later analyze to identify opportunities for improving the service contract.



Approach 3 – Service Oriented Architecture (part 02)

In other words, we describe an SOA where service execution reconciles the flexibility of LLMs for value creation with the rigor of service contracts for predictable and secure operation. The entire mechanism is monitored by a supervisory LLM that supports the continuous refinement of service contracts.

Conclusion

The analysis presented in this paper suggests that the hallucinatory tendencies of LLMs need not pose a barrier to large-scale automation, provided that their operation is framed by formal and deterministic service contracts.

The evidence indicates that realizing a sustainable return on investment from generative AI in enterprise contexts requires systematic monitoring of outputs, with oversight calibrated to the specificity of each use case. To this end, the role of the SOA architect becomes essential: structuring service contracts, identifying appropriate enabling technologies, establishing governance processes, and ensuring alignment with the overall data architecture.

Join Engage-Meta

On the community website, you'll find a collection of open-source resources that highlight the importance of working methodically to deploy AI at scale. Of course, this requires an effort to grasp the underlying concepts, and it's often less immediately rewarding than jumping straight into using NoCode-AI tools.

But taking the time to read, understand, and formalize complex thinking is a strategic asset for gaining deeper control over AI. By getting involved with Engage-Meta, you achieve two goals at once: you develop a better understanding of the complexity of AI–data systems, while strengthening your ability to read, structure, and share your thinking with your teams.