Hexagonal Architecture Is Not Just a Development Pattern. It Is a Structural Condition of Governability.

Why Regulated AI Systems That Do Not Separate Their Domain from Their Infrastructure Struggle to Make Their Reliability Governable by Design

Twingital Institute / Jérôme Vetillard / April 2026

Introduction

The three preceding articles in this series have defended, from different angles, the same core idea.

First point: AI governance is not primarily a policy. It is an architecture. The properties of traceability, bounding, and separation of decision regimes must be constitutive of the system, not added around it.

Second point: measured performance is not operational reliability. Calibration, the applicability domain, and a representative validation protocol do not belong to a late refinement of the pipeline. They must be considered before the first training run.

Third point: event-driven architecture makes native traceability possible, because it inscribes the system's transitions in a structural memory rather than in instrumentation added after the fact.

These three theses share an implicit premise that now needs to be made explicit. It must be possible, at least in principle, to define the decision logic of an AI system independently of the concrete modalities of its connection to the world. If this separation is not ensured, then the system's governability remains dependent on local development conventions, the testability of the decision core stays partial, and reliability properties can be modified without the system making that modification visible at the level where it becomes normatively significant.

This premise is far from trivial. A large part of the AI systems deployed today are built in such a way that this independence remains partial, fragile, or even impossible to establish rigorously. Not from ill will. From architectural default.

Hexagonal architecture, formalized by Alistair Cockburn under the name *ports and adapters*, provides one of the most operational formulations of this separation. Its proposal is simple: the business domain of an application should only interact with the outside world through abstract interfaces that it defines itself. The concrete implementations of these interfaces can evolve, be substituted, or be tested independently, without the domain logic being modified.

The thesis of this article is the following: in AI systems operating in regulated environments, a strict separation between domain and infrastructure is not a stylistic refinement of development. It is a structural condition of governability. Hexagonal architecture provides a particularly clear and operational expression of this. It makes it structurally possible to distinguish between what the system computes, what conditions the validity of use of that computation, and what the system is authorized to engage in a decision.

This thesis has an explicit domain of validity. It applies primarily to systems whose decision logic must be auditable, testable independently of its deployment environment, and modifiable without silent regression on reliability properties. It does not claim that hexagonal architecture is the only valid pattern for every software system. It maintains that in regulated environments, no governability that is structurally testable, durable, and opposable is possible without an explicit, rigorous, and verifiable separation between what the system decides and how it interfaces with the world.

I. Terminological Clarification

Hexagonal architecture is frequently cited, rarely defined with precision, and often confused with neighboring patterns. Before making it an argument for regulated AI systems, the terms need to be clarified.

The *domain* is the core of the system. It comprises the business rules, decision policies, invariants, and constraints that define what the system does, independently of how it does it. In a toxicological scoring system, the domain includes the logic for qualifying the applicability domain, the rules for conditional presentation of a score, the escalation policy toward a human expert, and the invariants according to which a calibrated score may or may not be communicated. In a clinical system, it includes the policies for exposing or not exposing a result, the escalation thresholds, the audit constraints, and the differentiated decision regimes depending on the context of use. In both cases, the domain does not know whether it is called via a REST API, a command-line interface, a unit test, or a nightly batch.

A *port* is an abstract interface defined by the domain to express a dependency. The domain states what it needs, in its own terms, without depending on any particular technology. It may need to calibrate a score, qualify an input relative to its validity space, record an inference, or route a request to a specialized predictor. This need is formulated as a contract, not as an implementation.

An *adapter* is the concrete implementation of that contract. It translates a real interaction from the outside world or from the infrastructure into a call intelligible to the domain, or vice versa. Adapters depend on the domain. The domain does not depend on adapters.

The fundamental distinction is therefore the following: *what the system does* must be separated from *how it does it*, and this separation must be guaranteed structurally, not by team discipline, documentation, or good intentions.

This architecture differs from the classic layered architecture, where business logic can still depend on persistence or a framework, through dependency inversion: it is the outside that connects to the domain, not the other way around. It also differs from Clean Architecture and Onion Architecture, with which it shares the dependency inversion principle, in what it places at the foreground: not the sole stratification of layers, but the question of boundaries and interaction contracts. This is precisely what makes it particularly fruitful for thinking about the governability of high-stakes AI systems.

II. Why Regulated AI Systems Remain Structurally Coupled

Most machine learning pipelines are not designed, originally, as governable systems. They are designed as trajectories of experimentation that progressively become deployable.

The pattern is familiar. A notebook becomes a script. The script becomes a service. The service is exposed via an API. The preprocessing, inference, logging, feature retrieval, possible calibration, and score presentation steps then become mixed in the same execution flow, sometimes in the same file, often in the same objects. The decision logic exists, but it does not exist as an isolable domain. It is dispersed across a chain of functions that simultaneously know what they compute and how they compute it.

This is not a moral failing, nor always a competence failing. It is the historical result of an ML development culture centered on rapid iteration, exploratory experimentation, and incremental production migration of prototypes. When moving from model.predict(X) to an inference endpoint, one often encapsulates the prototype instead of rearchitecting it.

In regulated environments, this coupling produces at least three structural insufficiencies.

The first is the impossibility of testing the domain in isolation. If the logic for qualifying the applicability domain depends directly on a feature store or a concrete implementation, testing it already requires part of the infrastructure. Regressions on reliability properties become harder to detect early, and their discovery slides toward late integration or production behavior.

The second is the impossibility of substituting implementations without touching the decision logic. Moving from isotonic calibration to Platt scaling for a given endpoint should be treatable as an implementation change, not a rewrite of the decision core.

When it is not, every change of tooling or method becomes a diffuse risk of regression on the system's behavior.

The third is the impossibility of making certain properties genuinely constitutive. In a coupled system, traceability, local validity qualification, or the decision not to display a score can always be implemented in the flow. They exist, but as code among other code. They can be forgotten, bypassed, or degraded during a refactoring without the system making it explicitly visible that a critical property of its reliability regime has just been touched.

The central problem is therefore less a software style problem than a problem of structural assignment of responsibility. As long as the system does not clearly separate the domain from its concrete dependencies, it cannot render explicit what belongs to computation, what belongs to decision, and what belongs to the conditions of engagement of the output in the real world.

III. Reliability Ports : Original Contribution

This is where the central contribution of this article intervenes. I propose the concept of *reliability port*.

In an AI system built on a strict separation between domain and infrastructure, certain operational reliability properties can be defined as contracts of the domain itself. They then cease to be simple peripheral additions to the pipeline. They become normative dependencies without which the output cannot be legitimately engaged in a decision.

A reliability port is a contract of the domain on which depends not only the functioning of the system, but the legitimacy of use of its output in a decision.

This is therefore not a secondary port among others, useful to the general functioning of the system. It is a port whose absence, failure, or non-conformity modifies the very legitimacy of the output. It does not only serve to run the system. It conditions the terms under which the system is authorized to produce a usable response.

In a regulated AI system:

- the port of calibration when the score must be interpreted as a probability or readable confidence level,
- the port of applicability domain qualification when the system must know whether an input lies within a sufficient validity space,
- the port of constitutive traceability when each inference must be recorded in an opposable manner,

- and the port of decision policy when the decision to present, qualify, suspend, or escalate an output depends on several combined signals,

typically fall into this category.

This formalization has three major operational consequences.

First, the contract becomes relatively stable while the implementation can evolve. The domain can express that it needs a local validity qualification without knowing whether it is realized by k-NN, Mahalanobis distance, or density estimation. In an audit, it is then possible to show that the domain logic has not changed, even if the implementation has been improved.

Second, these ports are testable by injection. In testing, real implementations can be replaced by control adapters that simulate out-of-domain cases, degraded calibrations, logging failures, or particular confidence levels which makes it possible to test the system's policy in the face of these signals without reconstructing the entire infrastructure.

Third, their suppression becomes costly and visible. Removing a constitutive traceability mechanism or domain qualification is no longer a quiet omission in a refactoring. It is an explicit modification of the domain contract. The act becomes deliberate, traceable, and contestable.

This is where architecture ceases to be a question of modularity and becomes a technology of responsibility.

IV. Articulation with the Preceding Theses

Architectural governance. In the article on architectural governance, I argued that certain properties must be constitutive of the system: native traceability, operational qualification of the validity domain, structural separation of decision regimes. Reliability ports provide the technical translation. Traceability becomes a port, domain qualification becomes a port, the output decision policy becomes a port. The critical point remains the same: the fact that a model has produced a score does not mean the system is justified in communicating it in a decisionally exploitable form.

Measured performance versus operational reliability. The article on this distinction argued that calibration, applicability domain, and relevant validation protocol must be thought of as properties of the system, not as late refinements. The hexagon makes it possible to understand why this requirement can become structural. The isotonic calibration and applicability domain qualification mechanism in ToxTwin can be thought

of as adapters implementing reliability ports which makes it possible to evolve the implementation without rewriting the presentation and escalation logic.

Event-driven architecture. Event-driven architecture and hexagonal architecture do not address the same problem. The former concerns primarily the temporality, distribution, and memory of the system's transitions. The latter concerns the structure of dependencies and domain separation. They are therefore complementary rather than competing. In a system combining both, the event journal can be a secondary adapter of the traceability port. The hexagon makes the dependency substitutable and explicit; the EDA confers on the system a native memory of produced events. One organizes the space of responsibilities, the other the time of functioning.

V. What the Market Does Not Industrialize

The MLOps tooling market is increasingly industrializing the model lifecycle. It versions, deploys, compares, monitors, rolls back, observes. But it still leaves largely underspecified the architecture of the decision.

A pipeline managed by MLflow, Vertex AI, or SageMaker can remain entirely coupled from the standpoint of its domain logic. The tool will track metrics, artifacts, deployments, and even distribution drifts. It will not, however, guarantee that the system knows how to distinguish what belongs to raw computation, local validity qualification, decisional engagement of the output, or the obligation to escalate.

ML development frameworks offer effective abstractions for datasets, models, inference chains, and workflows. They do not, by themselves, impose a separation between domain and infrastructure in the sense that interests us here. They organize processing. They do not define the architecture of responsibility.

Regulatory frameworks, for their part, impose documentation, risk management, traceability, and post-market surveillance obligations. They do not impose any particular architectural pattern. A highly coupled system can be compliant. It will simply be harder to test, evolve, audit, and govern over time. In environments where lifecycles are long and controlled modification requirements are high, this difficulty is not a detail. It is a structural disadvantage.

VI. Two Implementation Terrains

The following cases do not have the value of general demonstration. They serve to test the framework against two distinct architectures.

PREDICARE / Sentinelle IA. In a clinical digital twin architecture, the logic for predicting a decompensation risk can be designed as a domain indifferent to its mode of invocation. The patient mobile app, the clinical dashboard, and the population surveillance batch are distinct primary adapters connected to the same input contract. The event state journal is a secondary adapter of the traceability port. What this instance illustrates is that a system designed to carry clinical decisions across multiple usage contexts gains considerably in legibility and substitutability when the domain is isolated. What it does not prove is that migrating to a hexagon is costless, nor that it resolves all clinical governance problems on its own.

ToxTwin. In ToxTwin, the dynamic selection of a predictor adapted to an endpoint, via the V2.4 tri-router, can be thought of as the implementation of a routing port defined by the domain. The domain expresses that it needs the best available predictor according to a given policy, without needing to know the precise technology being mobilized. This makes it possible, in principle, to evolve the routing table or introduce a specialized model for metal coordination complexes in V3.0 without altering the general decision logic. What this instance illustrates is that an industrial system can evolve its critical reliability components without confusing implementation evolution with domain redefinition. What it does not prove is that adapter substitutability exempts from non-regression testing across the entire routing table.

VII. Limits

A credible position exposes its own limits.

The hexagon does not correct a poorly defined domain. If business rules are erroneous, inconsistent, or poorly understood, isolating them properly does not make them correct. The architecture improves testability and substitutability; it does not guarantee the intrinsic quality of the business logic.

Migrating an existing system toward a strict separation between domain and infrastructure has a real cost. Extracting the domain, defining the ports, creating the adapters, and stabilizing the contracts represents a non-trivial investment. For systems with low decision stakes or short lifecycles, this cost may not be justified.

Third-party foundational models pose a boundary problem. The domain can call them via a port. The formal separation remains. But the adapter stays partially opaque: one can

test the port contract, simulate its behavior, control its usages; one cannot audit in depth the real implementation.

Finally, the hexagon does not guarantee the correctness of adapters. An erroneous implementation of calibration or domain qualification can formally respect the contract while producing incorrect behavior. Hexagonal architecture makes integration tests more locatable and more explicit; it does not replace them.

Conclusion

Hexagonal architecture is often presented as a sound software design practice, stemming from clean code culture and Domain-Driven Design. This presentation is correct, but insufficient for regulated AI systems.

In these systems, the question is not merely how to better organize code. The question is whether it is structurally possible to make visible the distinction between what the system computes, what conditions the validity of use of that computation, and what the system is authorized to engage in a decision. As long as this distinction is not guaranteed by architecture, governability remains partial, reliability remains exposed to implicit couplings, and traceability still depends too often on local conventions.

Hexagonal architecture is therefore not here an aesthetic of decoupling. It is a technology of responsibility. It makes it possible to render visible, testable, and substitutable the dependencies without which a model output cannot be legitimately engaged in a decision.

The market is getting better and better at industrializing models. It still structures imperfectly the conditions of their accountability.

A system that does not know what it does independently of how it does it can produce useful scores. It cannot guarantee the legitimate conditions of their use.