2/19/2026

The Missing Layer

The Hidden Risk in Modern Autonomous Systems

By David Forbes

When Systems Act Without Permission

Modern computing systems are increasingly autonomous. They initiate actions, reconfigure themselves, recover from failure, and adapt to changing conditions without direct human involvement. In many cases, this autonomy is no longer optional; it is required to meet the scale, performance, and reliability demands placed on modern infrastructure.

As these systems grow more capable, however, a foundational question has been quietly left unanswered — not by any single discipline, but by the industry as a whole:

Who decides whether a system is allowed to act at all?

This is not an argument about making autonomous systems more capable. It is an argument about redefining what autonomy itself should mean. It is about whether execution itself is permitted under the conditions in which the system finds itself. And that distinction matters more now than it ever has.

We have become exceptionally good at building systems that decide *what* to do. We have invested heavily in orchestration, optimization, learning, and recovery. What we have not built with the same rigor are systems that can reliably determine whether they are *authorized to act in the first place.*

That gap — between capability and permission — is no longer theoretical. It is architectural. And as autonomous and AI-driven systems continue to expand into critical domains, its absence is becoming increasingly difficult to justify.

But as systems have grown more capable, a fundamental question has been left largely unanswered:

Not what the system *can* do.
Not how efficiently it can do it.
But whether execution itself is permitted under current conditions.

This question is not philosophical. It is architectural. And its absence is becoming one of the most consequential gaps in modern system design.

Capability Has Replaced Permission

Most modern systems are built around capability. If a component can execute an operation and receives a signal to do so, execution is treated as the correct and expected outcome.

Layers of control exist—policies, orchestrators, schedulers, and guardrails—but they are typically designed to shape *how* actions occur, not to evaluate whether those actions should occur at all.

Execution is the default.
Refusal is the exception.

This paper is intentionally architectural and conceptual. It does not provide implementation detail, algorithms, thresholds, state machines, or operational procedures. The goal is to define the missing legitimacy boundary in modern autonomy—not to specify a mechanism for enforcing it.

This design assumption worked when systems were simpler and tightly scoped. It becomes dangerous when systems are autonomous, distributed, and interconnected—especially when they are expected to operate safely under degraded, uncertain, or adversarial conditions.

In most architectures today, once an action request enters the system, it is assumed to be legitimate. The system may optimize, delay, or reroute it—but rarely does it stop to ask whether acting is still authorized. Until authority is treated as distinct from control, autonomy will continue to be measured by what systems can do, rather than by whether they are justified in doing it.

The industry frequently treats *control* and *authority* as interchangeable concepts. They are not. Confusing the two has led to systems that are exquisitely managed, carefully orchestrated, and fundamentally unauthorized.

Control governs *how* actions are performed. It determines sequencing, allocation, timing, and coordination. Controllers decide which resources are used, schedulers decide when tasks execute, and orchestrators decide where workloads are placed. Control logic assumes that execution is valid and focuses on making that execution efficient and reliable.

Authority answers a different question entirely. Authority determines *whether* an action is permitted to occur at all.

A system may be fully under control and still lack legitimate authority to act. In fact, many modern systems behave exactly this way: they execute flawlessly according to their control logic while operating outside the conditions that originally justified execution. Control ensures correctness of behavior; authority ensures legitimacy of behavior. One does not imply the other.

This distinction is subtle because, during normal operation, control and authority often align. When systems are healthy, configurations are stable, and trust assumptions hold, control decisions appear sufficient. The problem emerges when conditions change.

Schedulers, controllers, and orchestrators are not designed to ask whether the system's current state still permits execution. They do not evaluate whether a degradation has occurred, whether trust boundaries have shifted, or whether operational context has invalidated prior authorization. They simply continue to do what they were designed to do: execute.

As a result, systems frequently behave "correctly" in precisely the wrong moments. They recover aggressively when recovery is no longer safe. They reconfigure dynamically when

isolation would be more appropriate. They continue acting based on historical authorization that no longer applies.

Failures rarely originate in steady-state operation. They arise during transitions — recovery, reconfiguration, degradation, or partial loss of trust — when assumptions break faster than control logic can adapt. It is during these moments that authority must be re-evaluated, not assumed.

Without an explicit authority boundary, systems have no mechanism to refuse execution. Action becomes the default outcome, and restraint must be layered on reactively, if at all. Control continues uninterrupted, even as legitimacy erodes beneath it.

The ability to refuse execution is not a failure mode. It is a missing capability. And without it, autonomy remains fundamentally incomplete.

Optimization Without Restraint

The last decade of system design has been defined by optimization. Faster responses, smarter decisions, automated recovery, and self-healing infrastructure have become baseline expectations rather than exceptional capabilities. Systems are increasingly evaluated by how efficiently they act, how quickly they adapt, and how little human involvement they require.

What has not kept pace is restraint.

Modern systems are exceptionally good at doing *something*. They are far less capable of determining when *the correct response is to do nothing*. Action is treated as progress, execution as success. In most architectures, inactivity is interpreted as failure rather than intent.

This bias toward action becomes especially dangerous as artificial intelligence and autonomous agents are introduced into the execution path. These systems are designed to produce outputs, generate actions, and pursue objectives. Their value is measured by responsiveness and throughput. But objective pursuit without explicit permission boundaries is not autonomy — it is assumption.

When execution authority is implicit rather than explicit, systems operate on historical trust. They assume that if execution was permitted before, it remains permitted now. They assume that upstream inputs are still valid, that configurations still imply authorization, and that the conditions under which an action was once justified have not materially changed.

These assumptions hold until they don’t.

Under stress — during failure, recovery, degradation, or partial loss of trust — systems continue to act with confidence precisely when confidence is least warranted. They recover aggressively when recovery itself introduces risk. They reconfigure dynamically when isolation would be safer. They escalate actions based on outdated legitimacy.

In these moments, systems do not fail noisily. They fail *correctly*. They behave exactly as designed, executing optimized responses under conditions where execution should no longer be allowed.

This is the point at which optimization turns from strength into liability. The more capable the system becomes, the greater the harm it can cause when it lacks the ability to refuse action. ***Intelligence accelerates execution; it does not confer judgment***.

Without an explicit mechanism for restraint, autonomy scales faster than legitimacy. And when systems cannot distinguish between what they are capable of doing and what they are permitted to do, the consequences are not theoretical. They are operational.

The Missing Architectural Layer

What is missing from most modern systems is not intelligence, control, or observability. Those capabilities are abundant, well-funded, and aggressively optimized.

What is missing is a **first-class execution authority layer**.

This layer does not decide what should be done. It does not plan, optimize, predict, or learn. It exists for a single, narrowly defined purpose: to determine whether execution itself is permitted *at the moment it is requested*.

In systems where this layer exists, action is no longer assumed. Execution becomes conditional. Permission becomes revocable. Refusal becomes a legitimate and necessary outcome rather than an error condition.

An execution authority layer evaluates the system's own state before allowing action to proceed. It evaluates locally verifiable conditions—health signals, operating posture, and integrity context—and treats all external inputs as requests rather than commands. Signals may be received, but they are not obeyed automatically. Authority is adjudicated, not inherited.

Crucially, this evaluation must occur within the system boundary**.** Authority cannot be safely outsourced without becoming trust by proxy. External sources may inform the decision, but they cannot replace it. A system that defers its authority to something outside itself no longer knows whether it is acting legitimately; it is merely complying.

This distinction is subtle, which is why it has been overlooked. But it is foundational. Without a clear execution authority boundary, systems have no reliable way to determine when conditions have changed sufficiently to invalidate prior permission. They continue to act based on assumptions that are no longer true.

The absence of this layer is not an implementation gap. It is an architectural omission. And as autonomy increases, that omission becomes harder to justify—and harder to ignore.

Authority Is Not Cognition

There is a strong temptation to frame this problem as an intelligence problem. When systems behave incorrectly, the instinct is to assume they were not smart enough, not adaptive enough, or not informed enough. That instinct is understandable—and wrong.

The missing layer does not require cognition, optimization, or learning. It does not benefit from prediction or inference. It requires discipline.

Execution authority is not about deciding *what* should happen. It is about deciding whether anything is allowed to happen at all. That decision does not emerge from intelligence; it emerges from constraint.

This layer operates on explicit, concrete signals. It observes the system's own condition through sensors, health indicators, operational flags, mode registers, and state variables. These inputs are not interpreted or debated. They are evaluated. No intent is inferred. No tradeoffs are weighed. No outcomes are optimized.

The question it answers is binary and immediate: *do current conditions permit execution, or do they not?*

This is why execution authority cannot be treated as an agent. Agents are designed to act. They pursue objectives, adapt to circumstances, and attempt to succeed. Authority does none of these things. It does not try. It does not strive. It does not improve.

It constrains.

A more accurate analogy is not a decision-maker but an interlock—a mechanism designed to prevent motion unless specific, verifiable conditions are met. Interlocks do not negotiate. They do not reason. They either allow operation or they do not.

This distinction matters because interlocks are trusted precisely for what they *do not* do. Their value is not measured by how often they permit action, but by how reliably they prevent unsafe ones. They are designed to fail closed, not fail creatively.

When authority is treated as intelligence, systems become clever at acting under uncertainty. When authority is treated as constraint, systems become safe by refusing action when legitimacy cannot be established.

That difference is not semantic. It is architectural.

Why This Gap Has Been Tolerated

The absence of explicit execution authority has persisted for practical reasons. It is easier to assume permission than to prove it. It is simpler to centralize control than to distribute authority evaluation. It is more convenient to treat denial as failure rather than as correctness.

But convenience does not scale.

Much of this tolerance stems not from malice or intent, but from complacency embedded in how systems are built and maintained. Assuming permission reduces immediate friction. It avoids hard questions. It allows systems to keep moving forward without requiring anyone to stop and ask whether forward motion is still justified.

In operational environments, this assumption is often reinforced by habit. Systems that continue to function are rarely questioned, even when they function under increasingly fragile conditions. Over time, implicit trust replaces explicit validation. Authorization becomes historical rather than current. Execution is permitted because it has always been permitted.

This is not an intelligence failure. It is an attentiveness failure.

It is easier to rely on configuration than to understand state. Easier to depend on automation than to examine whether automation is still appropriate. Easier to respond after the fact than to prevent action before legitimacy is established. In many organizations, denial is viewed as disruption, while unchecked execution is viewed as productivity.

That framing is backwards.

As systems become more autonomous, the cost of unauthorized action grows nonlinearly. The consequences of acting without permission compound faster than the benefits of acting quickly. The question shifts from **“did the system behave as designed?”** to **“was the system justified in acting at all?”**

That question cannot be answered retroactively. Logs, audits, and postmortems can explain what happened, but they cannot undo an action that should never have occurred. Justification must exist at the moment of execution, or it does not exist at all.

This is where authority reveals its true nature. **Authority is not the ability to choose; it is the obligation to refuse.** It exists precisely to say no when conditions are ambiguous, degraded, or invalid—especially when action would be easier.

Systems that lack this obligation are not failing due to insufficient intelligence. They are failing because no mechanism exists to demand restraint. And without restraint, autonomy becomes complacent motion rather than legitimate action.

The Liability Horizon

The risks created by implicit execution authority are not theoretical. They are already visible across multiple domains, often surfacing only after localized actions have propagated outward into systemic failure.

In automated financial systems, for example, well-documented incidents have shown how trading platforms executed massive volumes of transactions within minutes, operating exactly as configured, yet under conditions where execution should no longer have been permitted. The systems did not miscalculate. They did not malfunction. They continued acting because no mechanism existed to revoke execution authority once conditions

diverged. Local decisions, made at machine speed, propagated into market-wide instability faster than human intervention could respond.

Similar patterns appear in large-scale network infrastructure. Automated routing and configuration systems are designed to heal, optimize, and adapt. Yet numerous widespread outages over the past decade have been traced to changes that were valid in isolation but catastrophic in context. Configuration updates were applied correctly, automation pipelines executed flawlessly, and control logic behaved as intended. The failure occurred because authority was assumed rather than re-evaluated. The network did not ask whether it was still permitted to change; it simply did.

In safety-critical systems, the consequences of implicit authority are even more stark. Investigations into aviation automation failures have highlighted how control systems continued asserting corrective behavior long after the assumptions that authorized those actions had collapsed. These systems did not lack intelligence or responsiveness. They lacked a mechanism to reassess whether execution itself remained legitimate as conditions evolved. Local sensor inputs triggered persistent action, and the absence of an execution boundary allowed those actions to propagate beyond safe limits.

More recently, large-scale software deployment failures have demonstrated how execution authority is often granted once and assumed indefinitely. Automated update pipelines pushed changes across vast fleets because the pipeline itself was authorized, even as downstream conditions rendered execution unsafe. Systems lacked the ability to refuse execution locally, to pause propagation, or to isolate impact. A single authorized action cascaded into widespread service disruption, not because the update was malicious or malformed, but because no authority boundary existed at the moment of execution.

Across these domains—finance, networking, safety-critical control, and software operations—the pattern is consistent. Systems acted decisively and correctly according to their control logic. The failure emerged because legitimacy was never reassessed. Execution authority was historical rather than current.

Crucially, these failures are not always the result of action. Inaction plays an equally significant role. Systems that should have refused execution often lacked the means to do so. Operators assumed automation would “do the right thing,” while automation assumed permission still applied. Responsibility diffused into silence until consequences surfaced elsewhere.

This is why the risk posed by unrestrained autonomy extends far beyond the local system. Autonomous actions rarely remain contained. They initiate downstream processes, influence peer systems, and trigger secondary automation that trusts their outputs. When authority is implicit, these chains of trust propagate unchecked.

Control without authority scales failure faster than manual error ever could.

As systems grow more autonomous and interconnected, the cost of unauthorized action grows nonlinearly. The question shifts from ***“did the system behave as designed?” to “was the system justified in acting at all?”*** That question cannot be answered after the fact.

Logs and audits can explain what happened, but they cannot undo actions that should never have occurred.

This is why the absence of an execution authority layer is becoming visible now—not because technology has failed, but because responsibility has become unavoidable. Systems that cannot refuse execution when conditions demand it are not autonomous. They are brittle. And the more capable they become, the wider the impact when legitimacy is assumed rather than enforced.

Toward Permission-Aware Systems

A permission-aware system does not ask whether it *can* act. It asks whether it *should*.

That shift changes how autonomy itself is understood. Autonomy is no longer measured solely by capability or responsiveness, but by legitimacy. Authority becomes a runtime concern rather than a configuration artifact—something that must be evaluated continuously, not assumed indefinitely.

In such systems, refusal is not treated as failure. It is treated as correctness.

Execution is no longer the default outcome. It is the conditional result of an explicit boundary between decision generation and execution permission. Systems may generate actions, propose responses, and identify opportunities, but none of those activities imply the right to act.

That right must be established.

This separation introduces a clear architectural boundary: decisions may exist without execution, and execution may be denied without error. The system remains operational even when action is withheld. In fact, it is precisely this capacity for restraint that allows autonomy to scale safely.

That boundary—the point at which a system determines whether it is permitted to act at all—is the missing layer.

It is not an optimization. It is not an enhancement. It is a structural requirement that has been quietly deferred as systems grew more capable without becoming more legitimate.

Once this boundary is recognized, it becomes difficult to ignore. Systems either possess a way to evaluate execution authority at runtime, or they do not. There is no middle ground.

A Different Way to Think About Safety

Why This Perspective Matters

Safety is often framed as the prevention of incorrect actions. In autonomous systems, that framing is incomplete.

A system that acts perfectly under invalid authority is still unsafe.

As autonomy continues to expand, systems will increasingly be judged not by how intelligently they act, but by how reliably they know when not to. This is not a question of performance or sophistication. It is a question of legitimacy.

For decades, system design has focused on improving what machines can do. We have built increasingly capable decision engines, controllers, planners, and optimizers. In doing so, we quietly embedded an assumption: that the right to act, once granted, remains valid unless explicitly revoked.

That assumption no longer holds.

Modern systems operate in environments defined by change—shifting trust boundaries, degraded states, partial failures, evolving missions, and ambiguous inputs. In such environments, execution permission cannot be a static property. It must be evaluated continuously, locally, and explicitly.

This requires a different way of thinking about computing itself.

Not as a sequence of actions to be optimized, but as a series of moments in which action must be justified. Not as systems that are always trying to do more, but as systems that understand when restraint is the correct response.

The most resilient architectures of the next decade will not be those that act fastest or learn most aggressively. They will be the ones that can refuse execution cleanly, predictably, and correctly when conditions demand it.
It begins with recognizing what has been missing all along.

David Forbes is an independent systems architect focused on execution authority and permission-aware autonomy.

References

1. **Kirilenko, A., Kyle, A., Samadi, M., & Tuzun, T. (2017).**
   The Flash Crash: High-Frequency Trading in an Electronic Market.
   Journal of Finance, 72(3), 967–998.
   — Demonstrates how automated financial systems executed correctly under invalid market conditions, propagating instability due to implicit execution authority.
2. **United States Securities and Exchange Commission & Commodity Futures Trading Commission (2010).**
   Findings Regarding the Market Events of May 6, 2010.
   — Illustrates large-scale systemic impact arising from automated execution continuing after underlying assumptions had collapsed.
3. **Labovitz, C., Ahuja, A., Bose, A., & Jahanian, F. (2010).**
   Delayed Internet Routing Convergence.
   IEEE/ACM Transactions on Networking, 19(6), 1752–1764.
   — Highlights how autonomous routing systems can propagate failure during reconfiguration without reevaluating legitimacy of execution.

4. **Madory, D. (2021).**
   Facebook Outage: What Happened.
   Kentik Blog.
   — Public analysis of a global network outage caused by automated configuration actions executed without a runtime authority boundary.
5. **National Transportation Safety Board (2019).**
   Aircraft Accident Report: Boeing 737-8 (MAX).
   — Documents how automated control systems continued asserting behavior under invalid assumptions due to lack of authority reassessment.
6. **Google Cloud (2020).**
   Service Disruption Incident Report.
   — Example of automated deployment and recovery mechanisms propagating failure across large fleets when execution authority was implicitly assumed.

These references are cited to illustrate recurring architectural patterns rather than to analyze individual incidents.