Nick Clark · Published July 2, 2026 · PDF

How to Stop an AI Agent From Attempting Actions Beyond Its Capability

If your agent dispatches a task, waits, and then fails with a timeout, you learned too late that the action was never executable. This guide describes an architecture that decides whether an action can structurally exist before any execution plan is built, so out-of-envelope actions are refused as a first-class result instead of discovered through failure. It is an architecture disclosed in United States Patent Application 19/647,395 (not a shipping library), and it centers on the Capability Awareness inventive step.

What You Are Building

You are building a gate that sits in front of execution and answers one question before your agent commits to anything: can an executable form of this action exist here, right now, given what the substrate can actually do? If the answer is no, the agent does not attempt the action. It refuses, and the refusal is a structured, recorded result the rest of the system can route on.

This is the search-intent problem restated. Developers ask "how do I stop an AI agent from attempting actions beyond its capability" because they have watched an agent confidently dispatch a task to a node that lacks the model, the accelerator, the sensor,

or the jurisdiction the task required, and then absorb a failure that looked like an infrastructure error rather than what it was: an action that was never possible. You want the "not possible" to be known before the attempt.

Who needs this: anyone running agents across heterogeneous compute, anyone whose agents call out to substrates whose capabilities change under load, and anyone who has to explain to an auditor why an agent tried something it could not do. The architecture below comes from the Capability Awareness inventive step disclosed in United States Patent Application 19/647,395.

Why the Obvious Approaches Fall Short

The common approaches are not wrong so much as aimed at a different question. Understanding the gap matters before you build.

Resource-availability checks ask whether a node has enough memory, CPU, or bandwidth at dispatch time. That is a real and useful check, but the filed specification is explicit that resource availability is a necessary but insufficient condition for capability. A node can have ample memory and idle cores and still lack the instruction set, the accelerator type, the loaded model weights, or the physical actuator the action requires. Passing a resource check does not mean an executable form of the action can exist.

Static capability registries ask what a node was configured to do. The spec's account is that these hand-maintained lists do not capture the temporal dynamics of capability: envelopes shift as hardware is provisioned or deprovisioned, as models load and unload, as environmental conditions change, or as other agents consume shared substrate resources. A registry that was accurate yesterday can quietly stop being accurate.

Permission and access control ask whether the action is allowed. This is genuinely a different axis. The specification draws a sharp line: permission, authorization, and access control all answer whether an operation is allowed, while capability answers whether an operation can structurally exist. An agent can be fully authorized and still structurally incapable. Conflating the two means an authorized-but-incapable action sails through the permission layer and fails downstream.

The structural gap under all three: the inability to execute is treated as a failure requiring retries, error handling, or escalation, rather than as a valid computational result that should inform routing, deferral, or decomposition. So the system builds a plan, dispatches it, and learns at runtime what it could have determined up front. Retrying an action on a structurally incapable substrate is the pathological case: no amount of waiting produces a capability the substrate does not have.

The Architecture

The disclosed approach makes capability a first-class computational state and evaluates it before any execution plan is constructed. Every mechanism below traces to the filing.

Capability as a computed determination, not a score. In the disclosure, capability is a structural condition describing whether an executable form of a given action can exist on a given substrate. It is not a metric, a probability, or a heuristic. It resolves to one of a bounded set of determinate outcomes: structurally possible, structurally impossible, structurally deferred, or must be rerouted to an alternative substrate. Critically, none of these is an error or a timeout. "Impossible" is a valid answer, not a fault.

Pre-synthesis ordering, enforced architecturally. The system does not build an execution plan and then check it. It first determines whether any executable form of the action can exist on the candidate substrate, and only if that determination resolves

affirmatively does it proceed to execution synthesis. This ordering is where "stop the agent from attempting" actually lives: the attempt is never constructed for an impossible action.

The capability envelope. Each substrate advertises a capability envelope, a structured description of its affordances across defined dimensions. The spec lists, at least: compute class (processor type such as CPU, GPU, TPU, FPGA, ASIC, or others; instruction set; word size; vectorization), memory architecture (addressable memory, bandwidth, cache topology, coherency, access patterns), model access (which inference models, knowledge bases, and embedding spaces are loaded or loadable), locality (geographic region, network position, latency, jurisdiction), execution guarantees (reliability, determinism, real-time scheduling, checkpoint-restore), and sensor and actuator interfaces (cameras, LiDAR, force-torque sensors, manipulators, and so on). The envelope is a living object: it updates as hardware, models, network conditions, or shared-resource contention change. The system does not rely on stale or statically configured capability data.

Dimension-by-dimension matching with three outcomes, not two. To decide, the system extracts the action's requirements into a structured requirements vector, retrieves the substrate's current envelope, and compares each dimension. Each dimension resolves to satisfied, unsatisfied (a shortfall that deferral or reconfiguration cannot fix), or conditionally satisfiable (currently short, but reachable through temporal deferral, reconfiguration, or partial decomposition). The aggregate composes from these: all satisfied gives structurally possible; an unsatisfiable dimension with no conditional path gives structurally impossible; conditional dimensions within a bounded horizon give structurally deferred; a dimension unsatisfied here but satisfied on a known alternative gives rerouted.

Capability held separate from permission. The two subsystems have no bidirectional dependency in the disclosure. The capability subsystem does not consult governance policy, and governance does not consult the envelope. They produce

independent determinations combined at a joint evaluation gate where both must hold for synthesis to proceed. This is what keeps an authorized-but-incapable action from slipping through, and it keeps capability evaluation from being contaminated by how privileged the agent happens to be.

Refusal as a recorded result: non-synthesis. When the joint capability-time-uncertainty evaluation says an executable form cannot or should not exist, the outcome is non-synthesis, and the disclosure treats it as a positive determination handled with the same rigor as success. It produces a structured non-synthesis record: the evaluated substrate, the action, the specific unsatisfied dimensions, the unmet temporal conditions, the uncertainty that exceeded threshold, and whether non-synthesis is permanent, temporal, conditional, or indeterminate. This record persists in the agent's lineage and is available to governance auditors, routing algorithms, and the agent's own planning. Refusal here is not a dead end; it is information the agent uses to reroute, defer, decompose, or revise the action.

Time and uncertainty travel with capability. The determination is jointly conditioned on three dimensions, not capability alone. Temporal executability forecasting projects each envelope dimension forward and finds the intersection window during which all required dimensions are simultaneously satisfied, distinguishing immediate executability, deferred executability, and temporal impossibility so the agent does not defer forever on a substrate that will never become capable. Uncertainty is a first-class propagated variable: it is the infrastructure's confidence in its own assessment, it accumulates as it passes through stages, and synthesis can be withheld when aggregate uncertainty exceeds a configured threshold.

Negotiating the substrate instead of just failing. When a dimension is conditionally satisfiable, the disclosed capability envelope negotiation lets the substrate advertise the modifications it could make (load a required model, provision a GPU partition, allocate reserved memory, activate a dormant sensor) with the estimated time and resource cost of each. The agent weighs that cost against rerouting to a substrate

that already satisfies the requirement. If it proceeds, it issues a capability acquisition plan subject to governance approval; the envelope updates and the determination is re-evaluated. Separately, a governed substrate resource negotiation runs before execution, where agents negotiate processor, memory, bandwidth, and sensor allocations as governed mutations that produce binding commitments the envelope folds into its executability determination.

How to Approach the Build

You are implementing this yourself. The steps below follow the architecture's own ordering.

1. **Model the envelope as data, per substrate.** Define a structured envelope with the dimensions your deployment actually gates on. Start with the spec's set (compute class, memory architecture, model access, locality, execution guarantees, sensor and actuator interfaces) and drop the ones irrelevant to you. Give each dimension defined semantics, value ranges, and comparison operators so matching can be formal rather than ad hoc.
2. **Keep the envelope live.** Wire the envelope to update on the events that change it: hardware provision/deprovision, model load/unload, network shifts, and shared-resource contention. A stale envelope reintroduces exactly the registry failure you are trying to escape.
3. **Extract requirements before you plan.** For each action, derive a requirements vector from the agent's intent and context that names, per dimension, the minimum substrate characteristics the action needs. This is the input to the gate.
4. **Implement the three-valued match and the aggregation rule.** Compare requirements against the envelope dimension by dimension, yielding satisfied / unsatisfied / conditionally satisfiable. Then compose the aggregate into structurally possible / impossible / deferred / rerouted. Record which dimensions were unsatisfied; downstream routing and decomposition need that detail.

An illustrative interface sketch (not a library, and faithful to the disclosure's shape):

```
determine_executability(action, substrate) -> Determination
  requirements = extract_requirements(action)          // requirements vecto
  envelope     = substrate.current_envelope()          // live, not cached
  per_dim      = { d: match(requirements[d], envelope[d]) // satisfied |
                        for d in requirements }           //  unsatisfied |
                                                          //  conditionally_
  return aggregate(per_dim)  // POSSIBLE | IMPOSSIBLE | DEFERRED | REROUTED
```

5. **Put the gate before synthesis, and enforce that ordering.** Execution synthesis runs only when the determination is possible (or has transitioned from deferred to possible), the temporal window is open, and aggregate uncertainty is below threshold. If any condition fails, do not synthesize. This is the single most important step; it is what stops the attempt.
6. **Make refusal structured.** On non-synthesis, emit the non-synthesis record with unsatisfied dimensions, unmet temporal conditions, exceeded uncertainty, and a permanent / temporal / conditional / indeterminate classification. Persist it. An agent that gets a structured refusal can act; one that gets an undifferentiated error cannot.
7. **Route on the refusal.** Use the recorded outcome to drive the agent's behavioral patterns from the disclosure: query multiple substrates and pick the best fit rather than the first; decompose into sub-objectives each routable to a covering substrate; defer when the forecast window fits the action's deadline; or, when nothing reachable covers the action, revise the action under policy-defined rules and record the revision.
8. **Keep permission independent.** Evaluate governance in a separate subsystem and combine at the gate so both capability and authorization must hold. Do not let one influence the other.

9. **Add negotiation and robustness once the core gate works.** Layer in envelope negotiation for conditionally satisfiable dimensions and governed resource negotiation for pre-execution commitments. Add the robustness paths: detect misreported capability through execution-time validation, handle partial failure by re-evaluating in-progress actions against a degraded envelope, and recalibrate forecasts when their accuracy drifts.

What This Does Not Give You

This is an architecture, not a drop-in library. There is no package to install and nothing here "just works" out of the box. You implement the envelope model, the matching and aggregation logic, the pre-synthesis gate, the records, and the routing yourself, against your own substrates and action taxonomy.

It is disclosed in a patent filing. It is not presented as a shipping product, and this guide reports no benchmarks, latency figures, or production results, because the filing does not state them and this guide will not invent them. Treat performance and overhead as things you must measure in your own deployment.

The gate is only as honest as the envelope. The disclosure itself calls out misreported capability, from bugs, undetected degradation, adversarial substrates, or stale data, and addresses it with execution-time validation and re-evaluation rather than assuming envelopes are always truthful. If you skip the live-update and validation work, a wrong envelope produces a wrong "possible," and the agent will attempt something it cannot do. The point of the architecture is to shrink that window, not to promise it never happens.

Finally, this addresses structural executability. It decides whether an action can exist on a substrate. It is not a correctness proof of the action's outcome, not a safety classifier for whether the action should be desired, and not a substitute for your governance policy, which the architecture deliberately keeps as a separate, independent axis.

Disclosure Scope

The approach described here is disclosed in United States Patent Application 19/647,395, which presents the Capability Awareness inventive step: a capability-envelope executability determination that gates permitted actions against an agent's real-time capability envelope, negotiates substrate resources, and refuses actions outside the envelope as a first-class result. This guide is educational. It explains an architecture so that a skilled developer can build it. It is not a warranty, not a specification of a released product, and not an offer of software, code, or an SDK. Any implementation, and its correctness, performance, and fitness for a purpose, is the reader's own responsibility.

Capability Awareness (/capability-awareness)

All 40 steps → (/inventive-steps)

Know what you can do before you try.

Chapter 6 (/patents/19-647395/chapters/capability)

PRIMARY TECHNICAL DISCLOSURE

- Capability-, Time-, and Uncertainty-Aware Execution in Autonomous Computational Networks (/articles/capability-time-and-uncertainty-aware-execution-in-autonomous-computational-networks)

SECONDARY TECHNICAL

- Capability as First-Class Computational State (/articles/capability-awareness/first-class-state)
- Capability Envelope for Substrates (/articles/capability-awareness/capability-envelope)
- Temporal Executability Forecasting (/articles/capability-awareness/temporal-forecasting)
- Uncertainty as First-Class Propagated Variable (/articles/capability-awareness/uncertainty-propagation)
- Capability Envelope Negotiation (/articles/capability-awareness/envelope-negotiation)
- Capability Genealogy Tracking (/articles/capability-awareness/genealogy-tracking)
- Biological Capability Extension (/articles/capability-awareness/biological-extension)

- Network-Level Capability Pressure (/articles/capability-awareness/network-pressure)
- Capability-Permission Distinction (/articles/capability-awareness/permission-distinction)
- Capability-Native Computation (/articles/capability-awareness/native-computation)
- Execution Synthesis (/articles/capability-awareness/execution-synthesis)
- Agent Behavior Under Constraints (/articles/capability-awareness/constrained-behavior)
- Predictive Network Planning Under Capability Pressure (/articles/capability-awareness/predictive-planning)
- Multi-Agent Contention Resolution (/articles/capability-awareness/contention-resolution)
- Capability Robustness Mechanisms (/articles/capability-awareness/robustness-mechanisms)
- Capability-Modulated Discovery Traversal (/articles/capability-awareness/discovery-constraint)
- Capability as Confidence Input (/articles/capability-awareness/confidence-input)
- Embodied Capability Envelopes (/articles/capability-awareness/embodied-envelopes)
- Substrate Resource Negotiation (/articles/capability-awareness/resource-negotiation)
- Place-Level Capability Envelope (/articles/capability-awareness/place-level-capability)
- Observation Staleness and TTL Governance (/articles/capability-awareness/observation-staleness-ttl)

APPLICATIONS · GENERAL

- Robotic Capability Assessment Before Commitment (/articles/capability-awareness/robotic-assessment)
- Edge Computing Resource Governance Through Capability Envelopes (/articles/capability-awareness/edge-resource-governance)
- Capability-Aware Surgical Robots: Refusing Procedures Beyond Calibrated Precision (/articles/capability-awareness/surgical-robotics)
- Capability-Aware Agricultural Robots: Terrain and Field-Condition Safety (/articles/capability-awareness/agricultural-robotics)
- Capability-Aware Autonomous Mining Equipment: Refusing Operations When Conditions Exceed the Safe Envelope (/articles/capability-awareness/mining-operations)
- Weather-Aware Autonomy for Offshore Energy Platforms (/articles/capability-awareness/offshore-energy)
- Capability Awareness for Warehouse Logistics Robotics (/articles/capability-awareness/warehouse-logistics)
- Capability Awareness for Construction Robotics (/articles/capability-awareness/construction-robotics)

- CORS and NTRIP RTK Without a Centralized Reference Network: Fleet-Emergent Precision Positioning (/articles/capability-awareness/cors-rtk-replacement)
- Self-Calibrating Autonomous Fleets: Precision Positioning Without Fixed Reference Infrastructure (/articles/capability-awareness/fleet-self-calibration)

APPLICATIONS · SPECIFIC

- Tesla FSD vs Capability-Aware Autonomy: What a Capability Envelope Adds (/articles/capability-awareness/tesla-fsd)
- John Deere Autonomous Tractors vs Capability-Governed Field Autonomy (/articles/capability-awareness/john-deere)
- KUKA Alternative: Capability-Aware Industrial Robots Beyond Static Parameters (/articles/capability-awareness/kuka)
- FANUC vs Capability-Aware Robot Execution (/articles/capability-awareness/fanuc)
- Universal Robots and Capability-Aware Cobots: Beyond Force Limiting (/articles/capability-awareness/universal-robots)
- ABB Robotics vs Capability-Aware Robot Execution (/articles/capability-awareness/abb-robotics)
- Yaskawa Motoman vs Capability-Aware Robot Execution (/articles/capability-awareness/yaskawa)
- Doosan Robotics Alternative: Governed Cobots With Capability Self-Knowledge (/articles/capability-awareness/doosan-robotics)
- Does Agility Robotics Digit Have Capability Awareness? (/articles/capability-awareness/agility-robotics)
- Figure AI Alternative: Governed Humanoid Execution With Capability Awareness (/articles/capability-awareness/figure-ai)
- Trimble RTK vs Capability-Aware Positioning Execution (/articles/capability-awareness/trimble-rtk)
- Hexagon SmartNet vs Capability-Aware Instrument Fleets (/articles/capability-awareness/hexagon-survey)
- Boston Dynamics (Spot / Atlas) vs a capability-envelope executability gate: how autonomy decides whether an action can structurally exist (/articles/capability-awareness/boston-dynamics)

Capability Awareness overview → (/capability-awareness)