Nick Clark · Published April 25, 2026 · PDF

How to Make an AI Agent Negotiate Resources Before Taking On a Task

If your agents dispatch work and then crash into resource-exhaustion errors, the problem is ordering: they commit before they know they can execute. This guide describes an architecture that gates every action against the agent's real-time capability envelope and negotiates substrate resources up front, so non-execution becomes a computed decision instead of a runtime failure. The approach is disclosed in United States Patent Application 19/647,395 (it is a design to build, not a shipping library), and it is the Capability Awareness inventive step.

What You Are Building

You are building an agent that decides whether it can actually execute a task, on a specific piece of infrastructure, before it commits to the task, and that bargains for the compute, memory, bandwidth, and device access it needs ahead of time. The searcher's intent is concrete: "my agent grabs a job, runs partway, then dies on out-of-memory or a missing model." The fix is not more retries. It is moving the feasibility question to the front and making resource acquisition an explicit, negotiated step.

This is a design pattern for anyone running agents across heterogeneous execution targets: a fleet of GPU nodes, mixed CPU and accelerator hosts, edge devices, or robots. If every node looks interchangeable until something breaks at runtime, this is the gap

you are closing. What follows describes an architecture disclosed in a patent filing. You implement it yourself.

Why the Obvious Approaches Fall Short

The common approach is dispatch-then-check. A scheduler sends a task to a node; the node either runs it or returns a failure. Feasibility is judged at dispatch time by resource-availability probes (enough memory, enough CPU, enough bandwidth) or by a static capability registry listing what each node supposedly does. Both are widely used and both are reasonable starting points. They fall short structurally, not because they are badly built.

First, resource availability is necessary but not sufficient. A node can have plenty of free memory and idle cores and still be unable to produce the work, because it lacks the architectural characteristic the task needs: the right accelerator type, the required model weights, a particular instruction set, a sensor, or an actuator. Free RAM does not conjure a missing model.

Second, static registries do not track time. A node's real capabilities shift as hardware is provisioned or removed, as models load and unload, as environmental conditions change, and as other agents consume shared resources. A registry written yesterday describes a node that may no longer exist.

Third, and most consequential, conventional systems treat "cannot execute" as an error to be caught, retried, or escalated. That collapses two very different situations into one failure code: a structural incapability (this node's architecture can never do this) and a transient shortage (this node could do it, just not right now). Retrying is correct for the second and pathological for the first, because no amount of waiting makes a structurally incapable node capable. Without the distinction, agents retry work on nodes that will never succeed.

The Architecture

The disclosed approach makes capability a first-class computational state, evaluated before any executable plan is constructed. Capability here is not a score or a probability. It is a computed determination of whether an executable form of the objective can exist on a candidate substrate, resolving to one of a bounded set of outcomes: structurally possible, structurally impossible, structurally deferred (possible later, within a forecasted window), or reroute to a different substrate. None of these is an error. Each is a valid result the agent can act on.

Capability is kept separate from permission. The spec draws a hard line between capability (can this operation structurally exist) and authorization (is this operation allowed). They are computed in architecturally separate subsystems with no cross-dependency, then combined at a single execution gate where both must be satisfied. This yields four quadrants; the important one is "authorized but not capable," which conventional systems mishandle by returning a generic error rather than routing, deferring, or decomposing.

Every substrate advertises a capability envelope. This is a structured, live description of a substrate's affordances along defined dimensions: compute class (CPU, GPU, TPU, FPGA, ASIC, and so on, plus instruction set and vectorization), memory architecture (addressable memory, bandwidth, coherency, access patterns), model access (which inference models, knowledge bases, and embeddings are loaded or loadable), locality (region, jurisdiction, latency, topology position), execution guarantees (reliability, determinism, checkpoint support), and sensor and actuator interfaces for embodied systems. The envelope is a dynamic object, updated as hardware, models, network conditions, or shared-resource consumption change.

Capability-native computation does the matching. The agent extracts a structured requirements vector from the objective, retrieves the substrate's current envelope, and compares dimension by dimension. Each dimension resolves to satisfied, unsatisfied, or conditionally satisfiable (currently short, but reachable through deferral,

reconfiguration, or decomposition). A composition rule aggregates these into the overall determination: all satisfied gives structurally possible; an unsatisfied dimension with no conditional path gives structurally impossible; conditional dimensions within a bounded horizon give structurally deferred; a dimension satisfied only on another known substrate gives reroute. Time and uncertainty are evaluated jointly with capability, so a node with the right hardware but no viable time window, or with excessive uncertainty in the assessment, does not proceed to synthesis.

Two distinct negotiation mechanisms sit on top of this:

Capability envelope negotiation handles the conditionally-satisfiable case. When a substrate almost matches, it can advertise the specific modifications it could make, for example loading a required model, provisioning a GPU partition, allocating reserved memory, or activating a dormant sensor, each with an estimated time and resource cost. The agent weighs that modification cost against rerouting to a substrate that already matches, factoring in the resulting temporal window (including the time to perform the modification), the opportunity cost of occupying the substrate during modification, and the agent's own urgency. If it proceeds, it issues a capability acquisition plan, a structured request subject to governance approval, the substrate modifies its envelope, and the determination is re-evaluated.

Governed substrate resource negotiation is the direct answer to the search query: agents negotiate with substrates and with other agents for processor allocation, memory budget, network bandwidth, and sensor access, before execution. It runs in three phases. A requirements declaration phase, where the initiating agent specifies the resources a planned execution needs, derived from the task's capability requirements and its envelope analysis. A counteroffer phase, where the substrate or competing agents respond with available allocations, alternative configurations, or conditional commitments naming the time windows when resources can be provided. And a commitment phase, where the parties reach a binding allocation that the capability envelope then folds into its executability determination for that planned execution.

Every offer, counteroffer, acceptance, and commitment is a governed mutation, evaluated and recorded in the agent's lineage, so the whole negotiation is auditable. Multi-party negotiation is supported, with each participant's offers bounded by its own envelope and policy (an agent cannot offer resources it does not control; a substrate cannot commit beyond its advertised envelope). Deadlocks resolve through policy-defined escalation to a higher authority, or by the agent's forecasting engine generating a lower-resource alternative plan.

Because commitments are secured before execution, synthesis proceeds against known-available resources. The spec also describes a pre-commitment validation that re-checks capability and the time window just before submitting the plan, aborting to non-synthesis if a resource that was available at synthesis start has since been consumed.

How to Approach the Build

The steps below are the order in which a developer would implement this. The interface sketches are illustrative and faithful to the spec, not a package you can install.

1. **Model the capability envelope as a first-class object per substrate.** Give each node a live, structured envelope across the dimensions above. Wire it to update on the events that actually change capability: hardware provision or deprovision, model load or unload, network shifts, and shared-resource consumption. Do not cache it statically.
2. **Define objective requirements as a comparable vector.** Extract, from each task, the minimum substrate characteristics per dimension. The point is a formal, dimension-by-dimension comparison, so requirements and envelopes must share semantics and comparison operators.

```
determine(objective, substrate) -> {
  per_dimension: [satisfied | unsatisfied | conditionally_satisfiable],
  aggregate: structurally_possible | impossible | deferred | reroute,
  uncertainty_bound, forecast_window?
}   // illustrative only
```

3. **Make the determination a gate before plan construction.** Enforce ordering at the architectural level: determine feasibility first; only synthesize an execution plan if the determination is affirmative. This is what removes the wasted work of building plans that cannot run.
4. **Keep capability and authorization in separate subsystems, combined at one gate.** Do not let a highly authorized agent look "more capable," or a highly capable substrate look "more authorized." Both conditions must pass the same execution gate independently.
5. **Add temporal forecasting and explicit uncertainty.** Project each envelope dimension forward using scheduled events, observed trends, and declared constraints, and compute the intersection window where all required dimensions are simultaneously satisfied. Carry bounded windows (earliest, latest, confidence), not point estimates, and propagate uncertainty so downstream decisions get more conservative as confidence drops.
6. **Implement the two negotiation protocols as governed mutation sequences.** For near-matches, build envelope negotiation (advertised modifications with costs, acquisition plan, re-evaluation). For up-front resource acquisition, build the three-phase governed resource negotiation (declare, counteroffer, commit), with binding commitments feeding back into the executability determination, per-participant policy bounds, and escalation on deadlock. Record every step in lineage.

7. **Treat non-execution as a structured result.** Emit a non-synthesis record naming the unsatisfied dimensions, unmet temporal conditions, exceeded uncertainty, and whether the condition is permanent, temporal, conditional, or indeterminate. Feed it to routing, deferral, decomposition, and objective-revision logic, so the agent does something informed instead of retrying blindly.

What This Does Not Give You

This is an architecture, not a drop-in library. There is no SDK to install and nothing here "just works" out of the box; you implement the envelopes, the matching, the forecasting, and the negotiation protocols yourself, and you supply the substrate integrations that report real capability state. The approach is disclosed in a patent filing. It has not been presented here as a benchmarked or production-proven product, and this guide states no performance numbers, because the spec states none.

Its usefulness depends on substrates that can honestly describe and modify their own envelopes and on a governance layer that can evaluate and record mutations; the spec assumes both. If your agents run in one homogeneous environment where every task is trivially executable, the machinery is overkill. And the guarantees are only as good as the inputs: forecasts carry uncertainty, envelopes can be misreported, and negotiations can deadlock. The design accounts for these as first-class conditions, but it does not make them disappear.

Disclosure Scope

The approach described here is disclosed in United States Patent Application 19/647,395. This guide is educational: it explains an architecture a developer can study and build, drawing solely on that filing's disclosure. It is not a warranty, a specification of any product, or an offer of software, and nothing in it should be read as a promise of fitness, availability, or results.

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)