Nick Clark · Published April 25, 2026 · PDF

How to Safely Push Field Updates to Autonomous Systems

If you operate a fleet of autonomous units in the field (vehicles, robots, drones, wearables) you eventually have to change their behavior after deployment: a new skill, a policy change, a firmware fix. Doing that safely, when connectivity is intermittent and a bad update can move a physical machine, is genuinely hard. This guide describes an architecture that treats each update as a signed, governed artifact that is validated against authority, policy, and continuity before it takes effect. The approach is disclosed in U.S. Provisional Application No. 64/049,409 and is the Spatial Adaptation inventive step. It is an architecture you build, not a library you install.

What You Are Building

You are building the delivery and admission path for changes that reach an autonomous unit after it has left the depot. Three kinds of change matter in practice: a governance-policy update (the rules the unit operates under), a firmware update (the code it runs), and a skill or model adaptation (a new capability layered onto the unit's existing behavior). All three share one uncomfortable property: when they land, they alter how a physical machine perceives or acts.

The goal is a pipeline where an update cannot silently take effect. Every update arrives as an artifact that carries who issued it, what it is allowed to change, and a cryptographic binding to its issuer. Before it activates, each receiving unit checks the artifact against its own authority model, against the policy state it already holds, and (for anything that touches behavior) against a sandbox. If any check fails, the update does not apply, and the unit can revert. This is the architecture described in U.S. Provisional Application No. 64/049,409.

Who needs this: anyone running units that act in the physical world and cannot assume a stable connection to a central update server at the moment an update is needed.

Why the Obvious Approaches Fall Short

The common approach is a centralized over-the-air (OTA) pipeline. A server signs an image, each device establishes a connection to that server, pulls the image, verifies a publisher signature, and installs. This works and is widely deployed. Its limits are structural, not a knock on any specific vendor:

- **It assumes connectivity at update time.** Each receiving device has to reach the distribution server. Units in tunnels, on ports, underground, or in sparse-coverage regions may not have that path when the update is needed.
- **A publisher signature answers "did this come from the publisher," not "is this publisher allowed to change this unit right now."** A valid signature does not encode hierarchical authority. Two validly signed updates from two different issuers are treated the same even if one issuer has no standing to govern this unit or this zone.
- **Consistency with the unit's current state is usually the installer's problem, handled ad hoc.** There is often no first-class check that the incoming policy is consistent with the policy the unit already holds, and no defined supersession rule when a higher authority and a lower authority disagree.

- **Skill and model updates frequently ship without pre-activation evaluation on the target.** The industry norm for adaptation layers is to activate them and observe. On a machine that actuates, "activate and observe" is a risky default.

The gap is not "we need better signatures." It is that the update needs to carry enough governed structure that the *receiving unit* can decide admission for itself, offline if necessary, rather than trusting the transport or a single publisher identity.

The Architecture

The core idea in the filing is that an update is not a blob you install; it is a **governed observation**: a self-describing, signed artifact that any receiving unit can evaluate on its own. Three mechanisms compose.

1. Updates propagate through the mesh, not through a central server. The filing discloses a mesh-distributed propagation mechanism in which a deploying authority publishes an update as a governed observation carrying the version, the content, the deploying authority's signature, and an applicability-scope specification identifying which devices are eligible. That observation propagates through the mesh by multi-hop relay and by mobile store-and-forward carriage, so an update emitted at an ingress point can be carried by units traversing a sparse-connectivity region and rebroadcast to devices there. No receiving device has to hold an out-of-band credential for a distribution server or establish a connection to one.

2. Each receiving unit admits the update through a composite admissibility evaluator. This is the safety gate. On receipt, the update is evaluated against three things the spec names explicitly: the authority credential of the deploying authority; the evidential weight that authority carries under the receiving unit's own authority taxonomy; and the consistency of the update with the unit's prior state. Only after that evaluation does the unit apply the change. Application is atomic: the unit transitions

from the prior policy state to the updated state as a unit, with a rollback mechanism that reverts to the prior state on admission failure or on receipt of a later revocation observation. Every transition is written to the unit's lineage record, so there is a deterministic trail of which version the unit was on and why it moved.

3. Anything that changes behavior gets a sandbox before it activates. The spec extends the same propagation pattern to firmware and to skill adapters, adding a pre-activation step. For firmware, the update is evaluated in a sandboxed execution environment on the receiving device to verify it does not violate governance-policy-defined safety, integrity, or capability-envelope properties; a firmware update that fails is not applied, and the failure plus the sandbox output is recorded. For skill and model adaptation, the filing describes a fuller sandbox: a sandboxed copy of the unit's cognitive substrate, representative evaluation contexts, a behavior evaluator that flags conflicts with artifacts already loaded, a governance summary of licensing, capability scope, dependency satisfaction, and authority compatibility, a risk projection, and an activation gate that returns admit-or-reject. The sandbox result becomes a certification record carried in the artifact, and it is itself admissible evidence to other units considering the same artifact.

For the adaptation case specifically, the artifact is a structured primitive (FIG. 8A in the filing) with named fields including an artifact identifier, an adaptation-technique identifier, the artifact content, a **capability scope** (the bounded range of contexts where it is allowed to apply), a compatibility specification, a licensing specification, a **dependency specification** (prerequisite artifacts), a **certification record** (sandbox results), a provenance-lineage record (training inputs, methodology, contributing authorities, antecedent artifacts), the issuing authority credential, a version identifier and version-lineage chain, and a cryptographic integrity attestation over all of it. That structure is what lets a unit reason about the update instead of merely trusting it.

The lifecycle closes the loop. The filing discloses publication, certification, active consumption, version-revision (each new version linked to its predecessor), deprecation, and retirement. Deprecation has graduated modes: soft-deprecation reduces the evidential weight given to the artifact, hard-deprecation rejects it after a transition window, and **security-driven deprecation rejects it immediately** on detection of a security-relevant defect. That immediate-reject path, plus the revocation-observation rollback, is your recall mechanism for a bad update already in the field.

How to Approach the Build

These are ordered steps to implement the architecture yourself. Treat the interface sketches below as illustrative shapes, not shipping code; they are faithful to the fields the filing names, and you write the real thing.

1. **Define the artifact envelope first.** Every update type shares one outer structure: issuer authority credential, applicability scope, version and version-lineage link, and an integrity attestation over the whole thing. Do not let policy, firmware, and skill updates diverge into three unrelated formats; the admission logic depends on their sharing this envelope.

```
UpdateArtifact {
  id, version, versionLineage[]      // links to prior version
  kind: policy | firmware | adaptation
  authorityCredential                // who issued it
  applicabilityScope                 // device class, geo, temporal
  content                            // policy body / image / adapter
  integrityAttestation               // signature binding all fields to t
}   // illustrative shape, not a spec API
```

2. **Build the authority model before the crypto.** The hard part is not signature verification; it is deciding *whose* signature counts, for what, and how a higher authority supersedes a lower one. Implement an authority taxonomy on the receiving unit that maps an issuer credential to an evidential weight and a permitted scope. The filing's supersession semantics say a higher-authority update supersedes a conflicting state set by a lower authority; encode that explicitly.
3. **Implement the admission evaluator as a hard gate, offline-capable.** It must run on the receiving unit with no server call. Minimum checks, all from the spec: verify the issuer credential; look up the issuer's evidential weight in the local taxonomy; and check consistency of the incoming update against the unit's current state. Reject on any failure. Do not make admission depend on reachability of the issuer.
4. **Make application atomic with a defined rollback point.** Snapshot the prior state, transition as a unit, and keep the prior state recoverable. Wire two triggers to rollback: admission failure and receipt of a later revocation observation for that artifact.
5. **Add the sandbox for behavior-changing updates.** For firmware, evaluate in isolation against your safety, integrity, and capability-envelope properties before applying. For adaptation artifacts, instantiate a sandboxed copy of the substrate, run representative contexts, check for conflicts with already-loaded artifacts, and gate activation on the result. Persist the outcome as a certification record on the artifact so peers can reuse it.
6. **Record lineage on every transition.** Version moved from, version moved to, the triggering event, and the admission or sandbox outcome. This is what makes an incident reconstructable and is called for throughout the filing.
7. **Wire the recall paths.** Implement the deprecation modes (soft, hard, security-driven immediate reject) and the revocation observation. You want to be able to pull a bad update from units already carrying it without a fleet-wide server round trip.

A pragmatic note: the mesh propagation and the admission gate are separable. You can adopt the signed-governed-artifact and local-admission model on top of a conventional transport first, then move propagation onto a mesh where connectivity demands it.

What This Does Not Give You

This is an architecture disclosed in a patent filing, not a drop-in library, an SDK, or a benchmarked product. There is nothing to `npm install`. You implement every component described here, and the design decisions the filing leaves open (which signature scheme, how the authority taxonomy is administered, how the sandbox models your specific substrate, what your safety and capability-envelope properties are) are yours to make and to validate.

It does not tell you *what* is safe for your machine. The sandbox and the admission gate are only as good as the governance-policy-defined safety, integrity, and capability-envelope properties you write; the architecture provides the structure to enforce those properties, not the properties themselves. It does not verify the correctness of an update's content, only that a credentialed authority within scope issued it and that it passed the checks you defined.

It also does not eliminate trust. It relocates trust from "the transport delivered this" to "an authority I recognize issued this within its scope." If your authority taxonomy is wrong, or an issuer's key is compromised, the gate faithfully admits a bad update. Revocation and security-driven deprecation are your response, and they only help once the defect is known.

Where it does not apply: a single connected device with a reliable link to one trusted server and no notion of competing authorities gets little from this over a well-run conventional OTA pipeline. The architecture earns its complexity when you have many units, intermittent connectivity, and more than one issuer whose standing differs by scope.

Disclosure Scope

The architecture described in this guide is disclosed in U.S. Provisional Application No. 64/049,409. Every mechanism described here (mesh-distributed propagation of updates as signed governed observations, admission through a composite admissibility evaluator against authority, evidential weight, and prior-state consistency, atomic application with rollback and revocation, sandbox pre-activation for firmware and adaptation artifacts, the adaptation-artifact field structure, and the lifecycle and deprecation modes) traces to that filing. This guide is educational. It is not a warranty, a specification, or an offer of software, and it does not describe a shipping or benchmarked product. You are responsible for your own implementation and for validating it against your own safety requirements.

Spatial Adaptation Artifacts (/spatial-adaptation)

All 40 steps → (/inventive-steps)

Runtime signed-skill loading. Sandbox-certified. Admissibility gate as skill router.

Provisional application

PRIMARY TECHNICAL DISCLOSURE

- Spatial Adaptation Artifacts: Runtime Skill Loading With Admissibility Gating (/articles/spatial-adaptation-artifacts-runtime-skill-loading-with-admissibility-gating)

SECONDARY TECHNICAL

- Runtime-Signed Adaptation Artifacts (/articles/spatial-adaptation/runtime-signed-artifacts)
- Sandbox Pre-Activation Certification (/articles/spatial-adaptation/sandbox-pre-activation-certification)
- Admissibility as Skill Router (/articles/spatial-adaptation/admissibility-as-skill-router)
- Always-Active Personal Layer (/articles/spatial-adaptation/always-active-personal-layer)

- Cascade Deactivation Dependencies (/articles/spatial-adaptation/cascade-deactivation-dependencies)
- Cross-Model Adaptation Portability (/articles/spatial-adaptation/cross-model-portability)
- Federated Skill Training (/articles/spatial-adaptation/federated-skill-training)
- Composite Licensing Intersection (/articles/spatial-adaptation/composite-licensing-intersection)
- Decentralized Mesh Adaptation Distribution (/articles/spatial-adaptation/decentralized-mesh-distribution)

APPLICATIONS · GENERAL

- Governed In-Field AI Model Adaptation for Defense Operations (/articles/spatial-adaptation/defense-field-adaptation)
- Safe Runtime Updates for AI-Driven Industrial Robots: Governed Adaptation Under ISO 10218 and the EU Machinery Regulation (/articles/spatial-adaptation/industrial-robotics-adaptive-update)
- Governing Adaptive Medical Device and SaMD Updates Under FDA PCCP and EU MDR (/articles/spatial-adaptation/medical-device-adaptive-update)
- Safe Rapid Security Updates for Safety-Critical Systems (/articles/spatial-adaptation/cybersecurity-rapid-update)
- Regulatory-Aware LLM Adaptation: Verifiable Governance for EU AI Act and FDA Compliance (/articles/spatial-adaptation/regulatory-aware-llm-adaptation)
- IEC 62304 Compliance for Continuously Adapting Medical Device Software (/articles/spatial-adaptation/iec-62304-medical-software)
- Enforcing NIST AI RMF Compliance at Runtime for AI Model Fleets (/articles/spatial-adaptation/nist-ai-rmf)
- UNECE R156 SUMS Compliance for In-Vehicle Software Updates (/articles/spatial-adaptation/unece-r156-software-update)

APPLICATIONS · SPECIFIC

- Governed Adaptation Beyond Anthropic Skills: Admissibility-Gated, Reversible Capability Loading (/articles/spatial-adaptation/anthropic-skills)
- OpenAI Fine-Tuning vs Governed Model Adaptation (/articles/spatial-adaptation/openai-fine-tuning)
- Tesla FSD Updates vs Governed Adaptation Artifacts (/articles/spatial-adaptation/tesla-fsd-updates)
- AWS Bedrock vs Governed Adaptation: The Certifiable-Artifact Layer (/articles/spatial-adaptation/aws-bedrock)
- Databricks Mosaic AI vs Governed Adaptation Artifacts (/articles/spatial-adaptation/databricks-ai)

- Google Vertex AI vs. Governed Adaptation: Who Owns Model-Adaptation Governance? (/articles/spatial-adaptation/google-vertex-ai)
- Hugging Face Hub Alternative for Governed Model Adaptation (/articles/spatial-adaptation/huggingface-hub)
- Governed Agent Adaptation vs Anthropic Claude MCP (/articles/spatial-adaptation/anthropic-claude-mcp)

Spatial Adaptation Artifacts overview → (/spatial-adaptation)