Nick Clark · Published April 25, 2026 · PDF

How to Safely Roll Back a Bad Update to an Autonomous Fleet

You pushed a model or policy update to a fleet of autonomous vehicles, robots, or field agents, and something is wrong in production. This guide teaches an architecture for making updates reversible by construction: governed, signed adaptation artifacts that are validated against policy and continuity before they take effect, and that revert cleanly when they misbehave. It describes an architecture disclosed in U.S. Provisional Application No. 64/049,409 (not a shipping library), organized around the Spatial Adaptation inventive step.

What You Are Building

You operate a fleet of autonomous units: delivery robots, warehouse vehicles, dock or airfield equipment, or dismounted agents carrying wearable devices. You ship them updates, whether a new model adaptation, a prompt or retrieval index, an expert-routing table, or a governance policy. One of those updates turns out to be bad: it degrades behavior, conflicts with something already loaded, or violates an operating constraint that only shows up in the field.

What you are building is the machinery that makes such an update reversible by construction. Not a manual firefight where you scramble to reflash units, but an architecture in which every update is a self-describing, signed artifact, in which

activation is gated on an evaluation the unit performs itself, and in which reverting to the last-known-good state is a defined operation that leaves an auditable trail. The goal is that "roll back the bad update" is a first-class capability of the system, not a recovery heroics story.

This is the problem the Spatial Adaptation inventive step addresses, as disclosed in the filing cited below. This guide teaches the approach; you implement it.

Why the Obvious Approaches Fall Short

The common way to ship fleet updates is a centralized over-the-air pipeline: a server holds the new firmware or model, each unit checks in, downloads it, and applies it. Rollback, when it exists, is a second push of the old image once someone notices trouble.

This works, and it is the right baseline for many systems. Its structural gaps show up specifically in autonomous fleets operating in the field:

- **The unit trusts the channel, not the artifact.** Authentication is typically a check that the download came from the expected server over TLS. Once the bytes arrive, they are applied. There is no per-artifact statement of *what authority stands behind this update* and *what it is allowed to change*, carried inside the artifact itself, that the unit re-evaluates against its own policy before activating.
- **Activation is a leap, not a preview.** The unit swaps in the new behavior and finds out in production whether it conflicts with what is already loaded. There is no standard step where the unit runs the candidate against representative situations on a copy of itself first.
- **Rollback is out-of-band and slow.** Reverting means a human notices, stages the old image, and pushes it back, over the same channel that may be exactly what is unavailable in a warehouse dead zone, a tunnel, or a port with intermittent connectivity.

- **There is no shared record of why.** After a bad update, reconstructing what changed, which units took it, and why it was reverted is a log-scraping exercise across machines.

None of this means OTA pipelines are wrong. It means that if you want *safe, reversible* fleet updates, reversibility has to be designed into the unit of change, not bolted on at the distribution layer.

The Architecture

The disclosed approach makes the *update itself* a governed object and makes *activation* a checkpoint. Every mechanism below traces to U.S. Provisional Application No. 64/049,409.

1. The update is a signed, self-describing adaptation artifact. In the filing, an update is packaged as an *adaptation artifact*: a structured object carrying an artifact identifier, the adaptation technique it uses, the adaptation content (parameter deltas, prompt templates, retrieval indices, or expert-routing tables), a *capability scope* stating the bounded range of contexts in which it is meant to apply, a compatibility specification for base substrates, a dependency specification naming prerequisite artifacts, a certification record, a provenance-lineage record of its training inputs and methodology, an *authority credential* identifying the issuing authority, a *version identifier and version-lineage chain*, and a *cryptographic integrity attestation* over all of the foregoing that binds the artifact to its issuing authority credential. The byte layout of this artifact is depicted in the filing's FIG. 8A. Because the artifact carries its own authority credential and signature, a receiving unit evaluates the update on its merits rather than trusting the transport.

2. Activation is gated by a composite admissibility evaluator. The filing discloses a *composite admissibility evaluator* as a single architectural primitive that each unit runs to decide whether to admit, gate, defer, reject, or escalate a governed

observation. The same evaluator that governs sensor observations and proposed actuations also governs *proposed adaptation-artifact activations*. It checks the artifact's authority credential against the unit's authority taxonomy, weighs the evidential trust assigned to that authority, and checks consistency with the unit's prior state before the artifact is allowed to take effect. Admission is a decision the unit makes, uniformly, using the same machinery it already uses for everything else it accepts.

3. Before activation, the artifact is sandbox-evaluated. Rather than swapping the new behavior straight into the operational substrate, the disclosed sandbox mechanism instantiates a *sandboxed copy* of the unit's cognitive substrate, applies the candidate artifact there, and runs it against representative generation contexts drawn from the unit's current task, standard evaluation suites, and its own stored observations. It produces a compatibility assessment against currently loaded artifacts, a performance preview, a governance summary of licensing and capability-scope boundaries and dependency satisfaction, and a risk projection, and only then does an activation-gate controller emit an admit-or-reject outcome. The result is written into the artifact's certification record and, per the filing, can be published as a governed certification observation that other units admit, so one unit's sandbox result informs the fleet.

4. Policy application is an atomic transition with a defined rollback. For governance-policy updates, the filing specifies that the receiving device *atomically transitions from the prior policy state to the updated policy state*, coupled with a *rollback mechanism enabling reversion to the prior policy state upon detection of admission failure or upon receipt of a subsequent revocation observation*. The prior state is not discarded on application; it remains the target of a defined revert. Firmware and skill-adapter updates follow the same propagation pattern with the sandbox-evaluation step added before application, and a firmware update that fails sandbox evaluation is not applied at all and is recorded together with the sandbox output.

5. Rollback to a prior version is a first-class operation. Because each artifact carries a version identifier and a version-lineage chain, the disclosed rollback mechanism lets a consuming unit *return to a prior-version adaptation artifact upon detection of post-activation governance-policy violations attributable to the current-version artifact*. The revert is not a guess about which old image to restage; it follows the version-lineage link. During generation itself, the routing mechanism inserts *generation checkpoints* at artifact-handoff transition points and *restores generation state* if a post-handoff policy violation is detected, so reversibility exists at the fine grain as well as the version grain.

6. Everything is recorded in lineage. Each of these events is written to a *lineage field*: the update event and the transition between policy versions, the sandbox-evaluation output, and, critically, the rollback event together with *the triggering violation and the restored artifact*. After a bad update, the record of what changed, why it was reverted, and what was restored is a property of the system, not a forensic reconstruction.

7. The fleet keeps prior versions on purpose. At the fleet scale, the filing discloses a *version-admissibility gate* that admits a new fleet-level artifact version only when governance-defined improvement thresholds are met against prior versions, and an *ecology-preservation* mechanism that deliberately retains prior-version adaptation artifacts across a plurality of agents for evolutionary redundancy. The last-known-good version is not merely archived off-fleet; it is kept live in part of the population, which is what makes a fleet-wide revert something you can actually fall back onto.

How to Approach the Build

You are implementing this yourself. A workable order:

1. **Define the artifact schema first.** Everything downstream depends on the update being self-describing. Give every update an identifier, a version and a link to its predecessor version, a capability scope, a dependency list, an issuing-authority credential, and a signature over the whole thing. The following is an *illustrative* sketch, not a library, and is faithful to the fields disclosed in the filing:

```
AdaptationArtifact {
  artifact_id
  technique                       // e.g. parameter-delta, prompt, retrieval
  content                         // the actual adaptation payload
  capability_scope                // where this is allowed to apply
  compatibility                   // which base substrates
  dependencies[]                  // prerequisite artifact ids + version ran
  certification_record            // sandbox-evaluation results
  provenance_lineage              // training inputs / methodology / antece
  authority_credential            // who stands behind this
  version_id, version_lineage[] // link to predecessor version(s)
  integrity_attestation           // signature over all of the above
}
```

2. **Stand up an authority model and verify signatures on the unit.** Decide which authorities may issue which classes of update, and have each unit verify the artifact's cryptographic integrity attestation and evaluate its authority credential *locally* before anything is applied. This is the step that lets you distrust the transport.
3. **Build the admission gate as one evaluator.** Route adaptation-artifact activations through the same admissibility decision path you use for other governed inputs: check authority, check evidential weight, check consistency with current state, emit admit / gate / defer / reject / escalate. Resist building a separate one-off code path just for updates.

4. **Insert the sandbox before activation.** Instantiate a copy of the unit's substrate, apply the candidate there, run it against representative contexts and against the currently loaded stack, and produce a compatibility and risk assessment. Gate real activation on that outcome, and write the outcome into the certification record so peers can reuse it.
5. **Make application atomic and keep the prior state.** Apply policy updates as an atomic transition that retains the prior policy state as an explicit revert target. Define admission-failure and revocation as triggers that flip you back to that prior state. For model and firmware updates, add the sandbox gate ahead of application.
6. **Implement version-lineage rollback.** On a post-activation violation attributable to the current version, revert along the version-lineage link to the prior version, rather than pushing a fresh image. Record the rollback, the triggering violation, and the restored artifact in lineage.
7. **Preserve last-known-good across the fleet.** Gate new fleet-wide versions on measured improvement over the prior version, and deliberately retain the prior version live in part of the population so the fleet always has a known-good state to fall back onto.
8. **Decide your propagation channel.** The filing distributes these governed artifacts through a mesh rather than requiring each unit to reach a central server, which is what extends coverage into sparse-connectivity regions. If your fleet has reliable connectivity, a conventional distribution channel can carry the same signed artifacts; the reversibility properties above live in the artifact and the unit, not in the channel.

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 artifact format, the signing and authority model, the admissibility evaluator, the sandbox, the atomic transition, and

the version-lineage rollback yourself, against your own substrate and safety requirements.

It 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, latencies, or guarantees, because the filing states none for these mechanisms and inventing them would be dishonest. Whether a revert is fast enough for your safety envelope, how large a sandbox evaluation you can afford on your hardware, and how much of the fleet you hold back on the prior version are engineering decisions you own.

The approach also assumes updates that fit the *governed adaptation artifact* model: bounded, self-describing modifications to a cognitive substrate or a policy state, each with a prior version to fall back to. It does not address failures outside that model, for example a hardware fault, a corrupted base substrate with no valid prior version, or a bad update whose harm is irreversible in the physical world before any software revert can take effect. Reversible software state is necessary but not sufficient for a safe fleet; the surrounding safety case is still yours to make.

Disclosure Scope

The architecture described in this guide, including the signed adaptation-artifact primitive, the composite admissibility evaluator gating activation, the pre-activation sandbox evaluation, the atomic policy-state transition with reversion, and the version-lineage rollback with lineage recording, is disclosed in U.S. Provisional Application No. 64/049,409, which anchors the Spatial Adaptation inventive step. This guide is educational: it teaches how to approach building such a system yourself. It is not a warranty, a specification of a shipping product, or an offer of software, and nothing here should be read as a guarantee of performance, safety, or fitness for any particular deployment.

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)