

POSITION PAPER · GOVERNANCE AS CODE

From Promise to Proof.

Policy as Code inside Governance as Code: how a rule stays declared, enforced, proven, and evidenced while the system around it keeps changing. Hybrid post-quantum transport as the worked example.

Every organization has a policy that says internal traffic is encrypted. Very few can show it, on demand, for the system as it is running today. A policy written as a document is a promise. A policy written as code can become proof, but only if the code keeps the rule true through every change that follows. This paper is the argument for treating a policy as something with a lifecycle, and for Governance as Code as the discipline that holds that lifecycle together.

PAPER

Position paper

FROM

Xiaotime Labs

AUDIENCE

CISO · Head of platform

A promise is not a control

Open any policy library and you will find sentences like this one: *all communication between internal services is encrypted in transit*. It is a good sentence. It maps to a control. It was approved by someone senior. And between one assessment and the next, nobody can say whether it is true.

That gap is not negligence. It is how documents work. A document describes an intent at the moment it was written. The system it describes changes every day: a new service ships, a library upgrades, a certificate expires, a configuration is copied from an old example. Each change is small and reasonable. Together they move the system away from the sentence, quietly, and the sentence does not notice.

The usual answer is to sample. Once a year someone checks a handful of connections and records that the policy held. That is evidence of a moment. It says nothing about the day after.

Continuous assurance is Everything as Code

It is tempting to say continuous assurance is Infrastructure as Code. Infrastructure as Code is the part most teams already have, so it gets the credit. But you can have Infrastructure as Code and still not know whether the running estate matches it. You can have it without a single rule checked when a change is proposed. You can have a pipeline that ships anything that compiles.

Continuous assurance takes all of it. **Infrastructure as Code** says what should exist. **Estate as Code** captures what actually exists, and shows the difference. **Governance as Code** says what is allowed, enforces it, and keeps the record. The pipeline is where those meet. Take any one away and the loop opens: infrastructure without the estate is a plan nobody checks, policy without a pipeline is a document, and a pipeline without policy is speed with no brakes.

Governance as Code is the newest of these disciplines, and the least understood. It is not one practice. It is four, on the same maturity curve the rest of Everything as Code already climbed: *policy* as code, *controls* as code, *evidence* as code, and *audit* as code. Policy as Code is the first of the four, and the one most people mean when they use the phrase. It is also the one that fails most quietly when it is done alone.

THE CLAIM

A policy governs only while it is **enforced, observed, and evidenced**.

Writing the rule in code is the first stage of its life, not the whole of it. Governance as Code is what keeps the rule true after it is written.

One rule, end to end

Abstractions about policy are easy to agree with and hard to act on, so here is one real rule, the one we hold our own platform to, followed through its whole life.

The rule: *every connection between internal services uses TLS 1.3 with hybrid post-quantum key exchange, and refuses a client that offers only classical key exchange*. Hybrid means the key is agreed with both a classical algorithm and ML-KEM, the post-quantum standard NIST published as FIPS 203. An attacker has to break both. Traffic recorded today is protected against later decryption by a quantum computer, and if the newer algorithm ever proves weak, the classical one still stands.

Staying on the internal network is not an exemption. Zero trust architecture says so plainly: communication is secured regardless of where it happens. So the rule covers the paths that never leave the building, which are exactly the paths a policy document tends to wave through.

STAGE ONE • POLICY AS CODE

Declared where the system is defined

The rule lives in the configuration of the component that terminates each connection: it offers the hybrid group and nothing else, and it requires TLS 1.3. There is no separate policy file that describes the configuration from the outside. The configuration *is* the policy, so the two cannot drift apart.

Each terminating component carries its own certificate, issued for its own name by the deployment's private authority, and nothing more. No component holds the key that issues certificates for the others. Every client trusts that private authority alone, not the public ones, so a certificate from anywhere else is refused.

STAGE TWO • POLICY AS CODE

Tested before anyone can weaken it

A test reads that configuration and fails if a classical group appears, if the minimum version drops, if a component starts serving a certificate it was not issued, or if a client is pointed at an unencrypted address. The test is written against the same file the system runs, so it cannot pass on a description while the system does something else.

STAGE THREE • CONTROLS AS CODE

Enforced at the change

The test is a required check in the pipeline. A change that would loosen the rule cannot merge, and the reviewer sees the reason in the failing check, not in a policy exception filed weeks later. Enforcement happens at the one moment a violation is cheapest: before it exists.

STAGE FOUR • CONTROLS AS CODE

Proven against the running system

Configuration and tests prove what the code says. They do not prove what the running system does. So a probe connects to each internal endpoint on a schedule, from inside the network, and records what it actually negotiated: the protocol version, the key exchange, and whether a client offering only classical key exchange was refused. Where a service has retired an older unencrypted port, the probe dials that port too, and expects to be turned away. A port that answers is a finding, not a footnote.

STAGE FIVE • EVIDENCE AND AUDIT AS CODE

Evidenced as a continuous record

Each observation lands in the risk register and is projected onto the controls it satisfies: transmission confidentiality and its cryptographic protection, the zero trust tenet, and the key exchange standard. An auditor reads a record that is as current as the last probe, linked back to the configuration and the change that produced it. There is no sample to schedule and no screenshot to collect. Collecting the evidence and running the system have become the same act.

All five stages express the same rule. Each catches something the others cannot: the configuration states it, the test protects it, the pipeline enforces it, the probe proves it, and the record shows it. Remove any one and you are back to trusting a promise.

Keeping a rule true is the hard part

Writing the rule took an afternoon. Keeping it true as the platform grew taught us more. These are the principles that came out of it, stated as design rules rather than war stories.

- ◆ **A check that cannot fail is not a check.** Each test counts what it inspected and fails when that count is zero, and a probe that examined nothing reports exactly that, never a pass. A matcher that silently stopped matching would otherwise report a clean result forever.
- ◆ **Absence of a finding is not a clean result.** A probe that only knows about the connections you have already secured will always report success. Coverage has to be stated beside the result, so a reader can see what was not examined as clearly as what passed.

- ◆ **The policy covers how things are deployed, not only how they are configured.** A correct configuration that the deployment never starts is a policy that exists only on paper. The step that brings each component up, and brings it back after a restart, is inside the rule's scope.
- ◆ **Renewal is part of the rule.** Certificates expire. A component that stops renewing falls out of compliance without anyone touching it. The probe reads each certificate's expiry and flags one the issuer should already have replaced.
- ◆ **Derive, do not enumerate.** A list of components maintained by hand goes stale the day after someone adds a service. Wherever possible, what the rule applies to is read from the system's own definition, so a new component is covered by construction.
- ◆ **Exceptions are records, not silence.** Where a component cannot meet the rule yet, the exception is written down with its reason and its date, and it stays visible until it closes.
- ◆ **Close the gap before you measure it more finely.** When the remaining gap is already known, the work is to close it. A sharper dashboard over a known gap is effort spent describing the problem instead of removing it.

THE SHIFT

Policy stops being something you **restate** and becomes something the system **carries**.

When the rule, its test, its enforcement, its proof, and its record all come from one source, nothing is copied, so nothing drifts. Change the rule once and every stage follows.

Why this is architecture, not a tool

Policy engines are a real advance. Engines such as Open Policy Agent let teams write rules in code and evaluate them automatically, and they belong in stage three. But an engine evaluates the request in front of it. It does not observe the running system, maintain its own coverage, notice an expired certificate, or produce evidence an auditor can follow. Those are the stages where policies quietly stop being true.

The difference is architectural. In a tool-shaped approach, the rule is restated in several places: the policy document, the engine's language, the monitoring rule, the evidence template. Each restatement is a copy, and copies drift. In the approach this paper argues for, the system's own configuration is the single source, and every stage reads from it. That is what Governance as Code means in practice. Governance is not a layer added beside the system. It is a property of how the system is defined, changed, and observed.

It is also why we apply it to ourselves first. We hold the platform's own internal transport to this rule, through the same pipeline we offer customers. A governance claim we do not run on our own systems is a claim we have not tested.

Framework alignment

Each stage maps to controls security and compliance teams already recognize.

STAGE	WHAT IT SATISFIES
Declared and tested	NIST SP 800-53 SC-8 and SC-8(1): transmission confidentiality, with cryptographic protection. NIST FIPS 203: ML-KEM for the post-quantum half of the key exchange.
Enforced at the change	NIST SP 800-53 CM-3: configuration change control, with the policy as a gate on every change.
Proven at runtime	NIST SP 800-53 CA-7 and NIST SP 800-137: continuous monitoring, observed rather than attested.
All internal paths	NIST SP 800-207: in a zero trust architecture, all communication is secured regardless of network location.
Evidenced	NIST CSF 2.0 Protect and Detect outcomes, supported by a continuous record rather than a periodic sample.

The objections, answered plainly

"We already have a policy engine."

Good. That is stage three, and it matters. The question is what proves stage four. If the only evidence that the rule holds in production is that the engine approved the deployment, you are trusting the plan, not observing the result.

"Probing internal traffic is overkill."

Internal is exactly where policy documents stop being checked, which is why it is where an attacker who gets inside looks first. The probe is also what turns the rule into evidence. Without it, the record shows that the configuration was correct when it merged, not that the system is correct now.

"Post-quantum is premature."

Data recorded today can be decrypted whenever a capable quantum computer exists, so long-lived secrets are exposed now, not later. Hybrid key exchange adds the post-quantum protection without giving up the classical protection you already trust. The cost is small, and modern TLS libraries support it.

"This only works on a greenfield system."

The rule was applied to a running platform, one internal path at a time, with the old unencrypted port left open until every caller had moved, and closed afterwards. Policy as Code is how a migration like that stays safe: each step is a change, each change passes the same gate, and the probe confirms each result.

Where to start

Pick one rule you already have in a document. It does not need to be cryptography. Walk it through the five stages. Where is it declared in the system itself? What test fails if someone weakens it? Does the pipeline block that change? What observes the running system and records the result? Who can read that record without asking you for a screenshot?

The first stage that has no answer is where your assurance stops. That is where to build next.

Read this as an argument. If it holds up against your environment, we should talk.

Related reading

- ◆ **Everything as Code.** The velocity case for moving governance into the code layer.
- ◆ **Estate as Code.** Capturing the whole estate, including the AI, as code you can govern.
- ◆ **Cryptography as Code.** What happened when we held our own infrastructure to our post-quantum advice.
- ◆ **Closing the Loop.** The continuous monitoring runtime that turns observations into evidence.
- ◆ **GRCDevSecOps.** Why governance belongs in the pipeline, not the report.
- ◆ **The Governed Fleet.** Why a fleet of AI agents is only as trustworthy as the control point they share.