Governance Stopped Us From Fixing the Wrong Problem

What a Git investigation taught me about executable governance, architectural intent, and controlled change

Justine Longla T. | JLT-LANE

Sometimes the most important engineering decision is deciding not to fix something yet.

During a recent repository-governance exercise, what initially looked like a straightforward Git cleanup problem turned into something much more interesting.

Several independently versioned repositories appeared inside a parent repository as Git links. The expected composition metadata was incomplete. Historical automation needed examination. Repository pointers had changed over time. There were old artifacts, quality gates, deployment considerations, and enough apparent disorder to make the obvious engineering response very tempting:

> ***Clean it up.***

But we didn't. And that turned out to be the right decision.

> ***The repositories weren't necessarily the problem. The relationship between them was insufficiently expressed.***

That distinction changed everything.

The Easy Fix Wasn't Yet an Authorized Fix

Engineers are trained to recognize undesirable state. A broken workflow? Fix it. Obsolete automation? Remove it. A malformed dependency? Correct it. A repository relationship that doesn't conform to the expected mechanism? Normalize it.

Those instincts are useful. But there is a dangerous leap between two very different questions:

> ***Does this look wrong?***

> ***Do we understand it well enough to change it?***

The second question requires evidence. So instead of immediately modifying the repository, we started investigating: What is the current state? What is committed? What is merely present locally? What changed? When did it change? Which commit introduced it? What was that commit supposed to accomplish? What else changed at the same time? Are other systems relying on this state? What does the documentation say these repositories own?

> ***What architectural intent was this structure trying to preserve?***

At that point, repository cleanup had become architectural investigation.

Then the Courtroom Appeared

Somewhere during the exercise, the technical process acquired characters. The jokes merely gave names to actors that already existed in the engineering system.

Caesar - execution asking for permission to act.

Security - enforcement and control.

Governance - authority derived from evidence.

The Clerk - evidence collection and preservation.

The Court - the quality gate.

Detective work - provenance and causal investigation.

The contract - declared expected behavior and state.

An objection - evidence contradicting an assumption.

Sustained - the objection satisfying the governing rule.

The verdict - the evaluated disposition.

The escort - controlled mutation after authorization.

.bak - apparently immortal archival evidence.

Napoleon - execution declaring 'Looks good enough - SHIP IT!' before governance has finished speaking.

Napoleon consequently spent increasing amounts of the investigation on Elba. The metaphor was entertaining, but underneath the jokes was a serious engineering model.

Governance Can Be Executable

Governance is often imagined as something that happens outside engineering: meetings, approval boards, policies, forms, and people deciding whether technical teams may proceed. Those mechanisms have their place. But mature engineering governance can also be made executable.

A contract declares expected state. A validation check compares observed state with that contract. Evidence contradicting an assumption raises an objection. A quality gate evaluates that objection. A failed condition withholds authority. A successful review permits a narrowly bounded mutation. The mutation itself can be constrained. Post-change verification determines whether the resulting state satisfies the contract.

Inspection -> Evidence -> Validation -> Authorization -> Controlled Mutation -> Verification

We deliberately refused to collapse that process into: Looks -> Fix it.

Nightly hygiene checks, state verification, provenance inspection, artifact review, quality gates, authorization boundaries, and post-change validation were not administrative ceremony around engineering. They were part of the engineering system.

Evidence Changed the Question

At first, several Git relationships appeared to be obvious cleanup candidates. Then provenance told a more complicated story.

The repositories were real. They had independent histories. Their documentation described distinct responsibilities across architecture, access control, delivery, observability, automation, and operations. The broader environment had intentionally evolved as a multi-repository platform engineering ecosystem.

We were no longer asking, 'Why are these strange repository objects here, and how do we remove them?' We were asking:

> ***What relationship were these repositories intended to represent?***
>
> ***Are these dependencies of the parent repository - or independently governed platform capabilities that should be composed through explicit contracts?***

That is no longer a Git-cleanup question. It is an architecture question.

Git could tell us what existed. History could tell us how it arrived. Documentation could provide evidence of ownership and intent. But those facts did not automatically authorize us to invent the future relationship. That required architecture.

Intent Is Not a Contract

> ***Something can be intentional and still be architecturally incomplete.***

We could establish that the repositories had been intentionally created. We could establish that they belonged to the same broader platform ecosystem. We could establish that Git had recorded repository pointers.

But none of that automatically established the architectural statement: 'This parent repository shall permanently consume these repositories through this composition mechanism.' That relationship requires a contract of its own.

> ***Intent explains why something may exist. A contract defines how it is allowed to participate in the system.***

Composition Is Not Containment

Two components belonging to the same platform does not mean one must physically live inside the other's repository.

A platform capability can have its own authoritative owner, repository, version, contract, tests, release lifecycle, and - where appropriate - deployment lifecycle.

A blueprint can declare what is required. A registry can identify approved implementations and versions. Contracts can establish compatibility and required inputs and outputs. Resolution can determine which components satisfy the blueprint. Assembly can compose them. Provisioning can establish the required target resources. Deployment can occur only where deployment is actually appropriate.

Surface -> Capability -> Component -> Contract -> Registry -> Blueprint -> Resolution -> Assembly -> Provisioning -> Deployment

Composition is an architectural relationship. Containment is merely a physical arrangement.

One Authority, Many Projections

As an ecosystem grows, websites, repositories, documentation, publications, and engineering surfaces often begin describing the same capability independently. Eventually, several versions of the same idea exist - and authority becomes unclear.

One capability, one authoritative owner; many surfaces may project it appropriately.

A commercial site might explain its value to a prospective client. An engineering surface might expose its implementation and composition model. Technical documentation might explain how to use it. A publication might explore the thinking behind it. A registry might expose machine-resolvable identity, version, provenance, compatibility, and contracts.

A surface presents a capability. It does not become another owner of that capability.

Governance Did More Than Say 'No'

Governance did not merely prevent execution. It repeatedly asked:

What evidence would make this decision knowable?

That question led us from symptoms to history, from history to provenance, from provenance to ownership, from ownership to architectural intent, and from architectural intent to target architecture.

Observed State -> Expected Contract -> Evidence -> Provenance -> Ownership Boundary -> Architectural Intent -> Target Architecture -> Authorized Disposition

That is not governance acting as a brake on engineering. Governance improved our understanding of the system.

Sometimes the correct verdict is permission to proceed. Sometimes it is rejection. And sometimes the most valuable verdict is: We now understand enough to know that we have been asking the wrong question.

Architecture Before Mutation

The purpose of governance is not to make engineering slower. It is to ensure that when execution finally receives authority, we are changing the right thing, for the right reason, within the right boundary, with evidence establishing what success means.

Sometimes governance authorizes execution. Sometimes it stops execution. And occasionally governance reveals that the proposed fix belongs to an architecture that has not yet been fully designed.

Don't mutate yet. Understand first.

Because one of the most expensive engineering mistakes is not failing to fix a problem. It is successfully implementing the wrong solution.

Governance stopped us from fixing the wrong problem.

Architecture first. Composition second. Mutation last.

JLT-LANE | Cloud Confidence. Delivered.