



BEYOND TOOLCHAINS

DevOps Is Not a Toolchain: Why the Wall Between Build and Run Must Fall

The longer leadership lesson behind faster software delivery in a cloud-first enterprise

DevOps Is Not a Toolchain: Why the Wall Between Build and Run Must Fall

Leadership · No. 74

Originally written August 2016 | Re-edited and re-rendered for the WhereBeyond Edition, 2025

"A deployment pipeline can shorten the journey to production, but it cannot make divided leaders share the consequences of arriving there."

1. The Friday boundary

At 4:17 on a Friday afternoon, a release manager asks four people whether a change can proceed. The development lead says the code passed every automated test. The infrastructure manager says the new virtual machines conform to the build standard. The service owner says the change window has been approved. The operations lead, looking at a dashboard that development cannot see, asks who will take the call if transaction times double at six o'clock.

Silence follows—not because nobody is competent, but because competence has been divided along the same boundary as responsibility. The people who built the change are measured on delivery. The people who will inherit it are measured on stability. The release manager is measured on adherence to process. Each has done what the organisation asked, yet the organisation cannot answer the simplest operational question: who owns the consequence?

This is the wall DevOps is meant to dismantle. In 2016, however, many enterprises are approaching DevOps as a procurement category. They buy a source-code repository, a build server, configuration automation and a deployment engine; they connect the boxes with scripts; they call the result transformation. Releases may indeed become quicker. The underlying division of labour remains untouched.

The longer view is less comfortable. DevOps is not primarily an improvement to the route by which software reaches production. It is a challenge to the way authority, risk and professional identity have been arranged around software for decades.

2. Why the toolchain story is so attractive

The toolchain interpretation succeeds because it offers visible motion without political disturbance. A new pipeline can be demonstrated. Deployment frequency can be counted. Provisioning that took ten days can be reduced to forty minutes. These are real improvements, and dismissing them would be foolish.

They are also administratively convenient. Capital can be approved, vendors assessed, environments built and staff trained without asking senior leaders to renegotiate who controls what. The transformation sits safely inside the technology function. Existing departments keep their mandates; existing committees keep their gates; existing measures keep their meaning.

Culture, by contrast, is difficult to place on a programme plan. It has no clean installation date. It exposes questions that tooling can postpone:

- May developers alter production configuration, and under what controls?
- Can operations reject a design before code is written, rather than after release paperwork arrives?
- Does a service owner control priorities across build and run expenditure?
- When a release fails, is the first act restoration and learning, or attribution and defence?
- Are teams rewarded for local performance or for the service experienced by the customer?

These are not questions about automation. They are questions about the constitution of the technology organisation.

A deployment pipeline can shorten the journey to production, but it cannot make divided leaders share the consequences of arriving there.

The attraction of tools is therefore not naïveté alone. It is an avoidance mechanism. Organisations can say they are adopting DevOps while preserving the very settlement DevOps calls into question.

3. The wall is made of rational behaviour

It is tempting to describe the division between development and operations as a failure of collaboration. That diagnosis is too soft. The wall persists because sensible people respond rationally to incompatible incentives.

Development is commonly organised around projects: defined scope, approved funding, a delivery date and a temporary team. Operations is organised around enduring services: availability, capacity, security, support cost and recoverability. One side is rewarded for introducing change; the other for controlling it. One side disperses when the project closes; the other lives with the residual complexity.

Consider a composite enterprise service that processes 180,000 customer instructions each day. A twelve-person development team spends nine months replacing a batch interface with a near-real-time service. The programme reports 96 per cent of scope complete and passes 1,240 automated tests. Two weeks before launch, operations receives the support model. It discovers 34 alerts, no agreed thresholds, a recovery procedure requiring three manual database steps, and an overnight dependency

supported by a different supplier.

The programme proposes a fortnight of “operational readiness”. Operations asks for six weeks. The argument is presented to the steering committee as resistance to change.

Yet the mechanism is plain. The development team has already incurred most of its cost and is judged against the launch date. Operations has received future liability with little influence over its design. Both estimates are shaped by where the consequences land. A better meeting will not reconcile that asymmetry. Shared accountability might.

When the service is finally launched, it suffers three priority incidents in ten days. Management responds by adding a production-readiness checklist and requiring another approval. The immediate reaction is understandable, but the new gate increases batch size: teams wait longer, assemble more changes and make each release harder to diagnose. A control intended to reduce risk quietly concentrates it.

This pattern recurs because organisations mistake friction for safety. The wall slows movement, so it appears to regulate movement. In reality it often delays information until the cost of acting on it is highest.

4. Speed does not create fragility; it reveals it

The serious anxiety behind resistance to DevOps is not cultural conservatism. It is that faster change will produce faster failure. In regulated sectors, where evidence, segregation of duties and predictable service matter, this concern deserves respect. An uncontrolled production estate with rapid deployment is not modern; it is merely dangerous at greater speed.

But the choice is not between deliberate control and reckless automation. It is between two ways of producing assurance.

The traditional way concentrates assurance in documents, hand-offs and episodic inspection. A change is prepared, reviewed and authorised as a package. This works tolerably where change is infrequent and systems are loosely coupled. As enterprises connect web channels, mobile services, shared data and cloud infrastructure, the package becomes harder to understand. The committee sees a summary; the system experiences the interaction.

The emerging alternative embeds evidence in the work itself. Tests are repeatable. Infrastructure definitions are versioned. Changes are small enough to trace. Deployment steps are automated. Logs and service measures are visible to the team that changed the system. Approval does not disappear, but it moves closer to the information required for judgement.

Question	Boundary-led model	Service-led model
Who owns delivery?	Project until handover	Team through operation
Where is assurance produced?	At gates and in documents	Continuously, with evidence at gates

Question	Boundary-led model	Service-led model
How is risk reduced?	Restrict the frequency of change	Reduce the size and uncertainty of change
When does operations influence design?	During readiness and acceptance	From the first design choices
What happens after failure?	Escalate across functions	Restore, learn and alter the system
What is optimised?	Departmental obligations	End-to-end service performance

The distinction matters because stability is not the absence of change. Stability is the ability to change without losing control. A service that can be altered only through heroic preparation is not stable; it is brittle.

The most revealing measures are therefore paired measures. Deployment frequency without failed-change rate rewards haste. Availability without time to restore can reward avoidance. Project milestones without support cost reward the transfer of unfinished work. The point is not to find one perfect metric, but to make local optimisation harder.

5. The strongest case for standardisation

Advocates of a tool-led approach have a stronger argument than its critics sometimes admit. Large enterprises cannot allow every team to invent its own delivery machinery. Fragmented tools multiply security exposures, support costs and scarce skills. A common pipeline can encode access controls, retain evidence, standardise environment creation and make good practice easier to repeat. In a mixed estate of packaged applications, bespoke systems and ageing platforms, standardisation is not bureaucracy by definition; it is often the precondition for scale.

Nor can culture substitute for engineering. Trust does not make a manual deployment repeatable. Shared purpose does not make an untested recovery procedure safe. The craft matters.

The error lies in reversing cause and effect. Standard tools can support a shared operating model, but they cannot create one. When imposed on an unchanged organisation, the common pipeline becomes another hand-off system: development submits a package, a central automation team maintains the scripts, operations approves the release, and service management receives the incident. The boxes are newer; the queue remains.

A useful standard creates a paved route, not a compulsory detour. It provides tested components, evidence and support while keeping the team close to the service. The test is not how many teams use the tool. The test is whether use of the tool shortens the distance between a decision and its operational consequence.

This is why the cultural and technical arguments should not be staged as rivals. The organisation needs both. Engineering without shared ownership accelerates the old

dysfunction. Shared ownership without engineering leaves goodwill trapped in manual work.

6. What leadership must surrender

DevOps is often delegated to engineers because its visible practices are technical. Its difficult work belongs to leadership because only leaders can change the boundaries within which engineers act.

The first surrender is the fiction that accountability can be transferred at a handover. Knowledge can be transferred imperfectly; accountability cannot. If a project team can meet its objectives by leaving an expensive or fragile service behind, the governance model has defined success too narrowly.

The second surrender is the comfort of functional measures. Leaders like measures that fit their organisation charts: delivery performance for development, availability for operations, compliance for risk, cost for infrastructure. Customers encounter none of these functions separately. They encounter a service. Leadership must therefore hold the service conversation above the departmental one.

The third surrender is the idea that control requires distance. Segregation of duties is important, particularly where a single individual must not initiate, approve and conceal a change. But segregation need not mean ignorance. Independent approval is stronger when the approver can inspect automated evidence, understand a small change and see its result. Distance often produces formal independence at the expense of practical insight.

The fourth surrender is the search for a cultural campaign. Posters about collaboration will not overcome funding that ends at go-live, targets that punish change, or incident reviews that begin by locating fault. Behaviour follows the operating environment with remarkable loyalty.

A credible leadership sequence is modest and concrete:

1. Choose one consequential service, not an enterprise-wide slogan.
2. Give a stable, cross-functional team responsibility from demand through operation.
3. Place operational requirements—monitoring, capacity, recovery, support and security—inside the definition of finished work.
4. Fund improvement and run activity alongside feature delivery, rather than treating them as separate claims on separate budgets.
5. Review paired service measures weekly: change volume and change failure, availability and restoration time, demand delivered and operational debt added.
6. Examine incidents as evidence about the system of work, while retaining clear accountability for negligence or concealment.

In one composite case, a payment service followed this sequence for a single release stream. Before the change, it released every six weeks, required eleven manual

approvals and took a median of 160 minutes to restore a failed deployment. The first aim was not daily release. It was to reduce uncertainty.

The team reduced each release from roughly 70 changes to fewer than 15, automated the seven repeatable checks, retained independent approval for higher-risk changes, and rehearsed rollback during working hours. After four months, the service released weekly. Failed changes fell from one in five releases to one in twelve, and median restoration time fell to 28 minutes. The decisive moment was not installation of the deployment engine. It came when the operations lead gained authority to prioritise recovery work in the same backlog as customer features—and the development lead joined the support rota for the first 48 hours after release.

Those mechanisms changed what people attended to. Smaller changes made evidence intelligible. Shared prioritisation made operational debt visible. Proximity to incidents altered design decisions before the next release. Culture shifted because the work and its consequences were reunited.

“The wall between development and operations survives less through hostility than through the careful distribution of consequences.”

7. The longer view

We should be wary of treating DevOps as a destination. The name may endure or fade; the underlying question will remain. As software becomes woven into more products, channels and regulatory obligations, can an organisation continue to separate those who create change from those who absorb its effects?

The answer is unlikely to be a universal team shape. A customer-facing digital service, a core transaction platform and a heavily outsourced package will not be governed identically. Scarce specialists will still need to work across services. Independent challenge will still matter. Some changes will still require formal windows and explicit authority. The aim is not organisational purity.

The aim is to preserve three properties despite those variations:

- Consequence stays visible to the maker. People who design and change a service see how it behaves in use.
- Operational knowledge arrives early. Reliability, recovery and support are design concerns, not acceptance tests.
- Authority follows responsibility. A team held to a service outcome can influence the priorities, funding and controls that determine it.

This is a more demanding ambition than continuous delivery. It asks leaders to see technology not as a sequence of projects feeding an operational machine, but as a set of

living services whose quality depends on repeated learning. It replaces the ceremonial transfer of risk with a continuous conversation about risk.

The organisations that grasp this will still invest in automation—heavily. But they will judge the investment by whether it changes the system of work. They will know that a faster pipeline laid across an old boundary may simply deliver unfinished consequences more efficiently.

DevOps, properly understood, is the refusal to let that boundary remain invisible. Its deepest promise is not speed. It is that the people with the knowledge to prevent failure, the authority to make trade-offs and the responsibility to live with the result can finally act as one system.