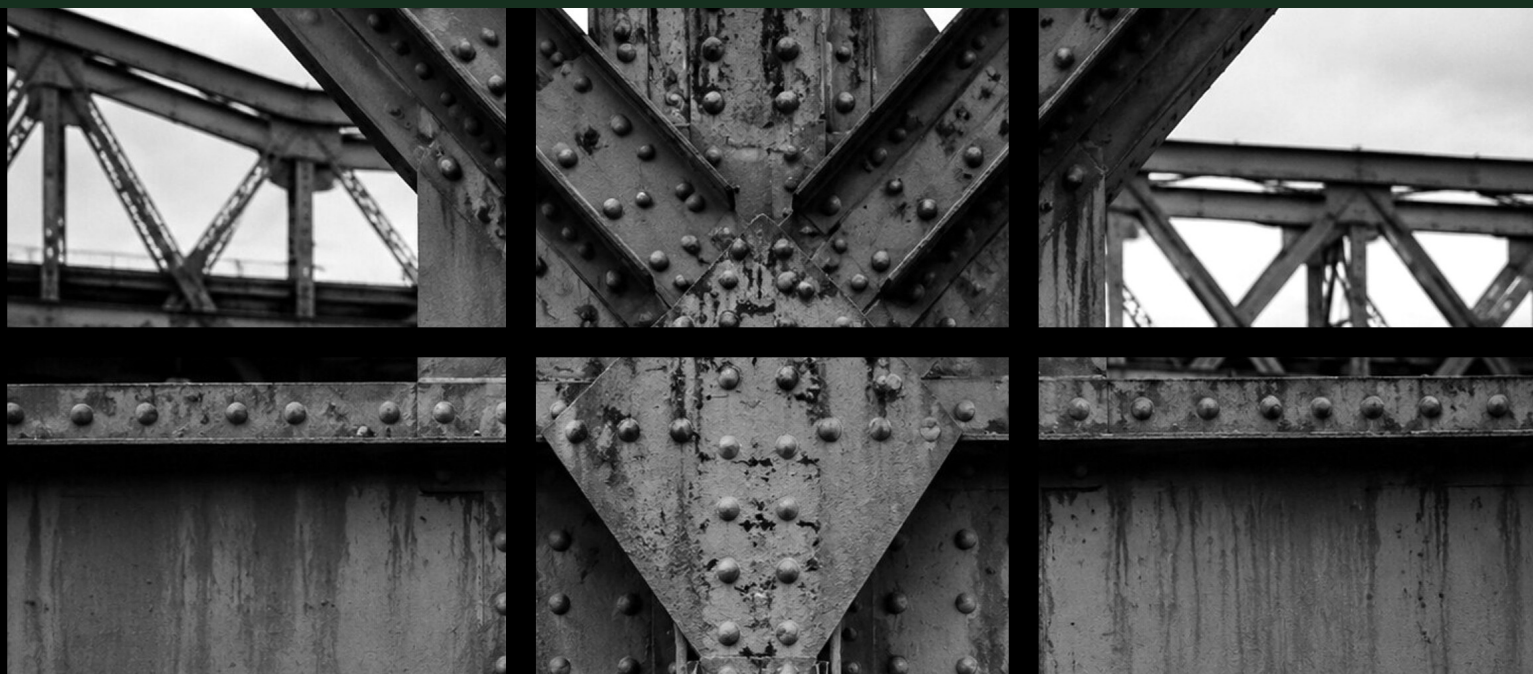


BEYOND THE TOOLCHAIN

Continuous Delivery Fails When Release Still Runs by Committee

Automation cannot create flow while production remains an exceptional event



Continuous Delivery Fails When Release Still Runs by Committee

Programme · No. 135

Originally written October 2010 | Re-edited and re-rendered for the WhereBeyond Edition, 2025

“The organisation does not reject continuous delivery directly; it makes each release expensive enough that batching becomes rational.”

The Release That Was Continuous Only Until Operations

At half past six on a Friday evening, a delivery team completed what it called a continuous-delivery release. The build server had compiled the code, run several thousand automated tests, assembled the package and placed it in the release directory. The same package had passed through development, system test and acceptance without being rebuilt.

Then the continuous flow stopped.

An operations manager opened a twelve-page implementation plan. A database administrator waited for permission to run two scripts. Four people joined a conference call to confirm that the change record had been approved. The business sponsor was told that, if anything went wrong after eight o'clock, the entire release would be backed out. By midnight, one small application change had consumed the attention of seventeen people.

This is the gap between continuous delivery as a concept and continuous delivery as an organisational reality. We are becoming much better at automating the movement of software. We remain far less willing to redesign the decisions, responsibilities and incentives surrounding that movement.

Automation Exposes the Real Constraint

The appeal of continuous delivery is easy to understand. Integration that once happened near the end of a project can happen with every change. Tests that once depended on a manual regression cycle can run repeatedly. A deployment package can be created in the same way every time. Errors become smaller, more visible and cheaper to correct.

These practices change the shape of delivery risk. A large, infrequent release concentrates uncertainty: many changes arrive together, the path to production is exercised rarely, and diagnosis becomes difficult because too much has moved at once. Smaller releases should reduce that uncertainty.

Yet the mechanism works only if the organisation allows smaller releases to remain small.

In practice, many organisations retain a release process designed for quarterly bundles. Every change, however modest, enters the same governance queue. The same evidence is requested. The same meeting must approve it. The same weekend window is reserved. Because the approval cost is high, teams combine changes to make the process

economical. The bundle grows, risk rises, and the organisation cites that risk as proof that the controls were necessary.

The organisation does not reject continuous delivery directly; it makes each release expensive enough that batching becomes rational.

That is why tooling alone so often disappoints. Faster builds do not create faster decisions. Automated tests do not clarify who owns production risk. A deployment script cannot resolve the tension between a project manager rewarded for meeting a date and an operations manager rewarded for avoiding disruption.

The Serious Objection

The strongest objection to continuous delivery is not resistance to change. It is that production systems are not laboratories.

Operations teams carry the consequences when a release fails at two in the morning. Auditors expect evidence that changes were authorised. Customer-facing systems may depend on ageing infrastructure, fragile interfaces and database changes that cannot be reversed simply. In a complex estate, a small application change can have effects beyond the team's line of sight. Under those conditions, independent review and controlled release windows are not bureaucratic habits; they are protections earned through painful experience.

That objection deserves respect. Some advocates speak as if smaller batches abolish operational risk. They do not. A badly understood dependency released frequently is still a badly understood dependency. An automated test suite can be extensive and still omit the failure that matters. A team confident in its application can remain ignorant of the estate around it.

But the conclusion should not be that every release requires the same ceremony. The better conclusion is that control must become proportionate, repeatable and based on evidence.

A low-impact change with automated regression results, a rehearsed deployment, clear monitoring and a tested back-out route should not travel through the same path as an irreversible database conversion. Treating them identically does not make the first safer. It consumes scarce attention that should be reserved for the second.

Where the Reality Breaks

Across early attempts, the same four fractures recur.

- Testing remains a project phase. Teams automate unit tests but still depend on a separate manual acceptance cycle at the end. The deployment machinery moves faster than the evidence required to trust it.
- Environments remain bespoke. Development and test can be prepared quickly, while production settings are maintained through documents and individual memory. The package is consistent; its destination is not.

- Authority remains functional. Development can approve code, testing can approve evidence and operations can approve release, but nobody owns the elapsed journey from change to usable service.
- Commercial arrangements reward artefacts. Suppliers are paid for milestones, documents and accepted releases rather than for maintaining a reliable flow of small changes.

A composite programme made this visible with a simple measure. Its automated build required eleven minutes. Deployment to the test environment required forty minutes. The average wait for the next production window was twenty-three days. Management spent three months debating whether the build should be reduced to eight minutes while the twenty-three-day queue remained untouched.

The figures were accurate. The priority was absurd.

Continuous Delivery Requires a Different Agreement

The real shift is not from manual deployment to automated deployment. It is from release as an exceptional event to release as a routine capability.

That requires an explicit agreement between development, testing, operations and the business.

1. Define classes of change. Separate routine, reversible changes from high-impact or irreversible ones. The evidence and authority required should follow the risk class.
2. Build evidence into the path. Test results, configuration checks, deployment records and back-out evidence should be produced by the process rather than assembled for a meeting.
3. Keep one package. The item tested should be the item released. Rebuilding for each environment destroys much of the confidence automation is meant to create.
4. Make operations part of design. Capacity, monitoring, recovery and support must be considered while the change is shaped, not after development declares it complete.
5. Measure waiting as rigorously as working. Build time matters, but approval time, environment time and release-window delay reveal whether the organisation is actually improving flow.

None of this removes accountability. It relocates accountability from a late ceremonial approval to the design of a trustworthy delivery system.

The Point We Are Missing

Continuous delivery is sometimes presented as the natural next step after continuous integration: improve the build, automate more tests, script deployment, then release more often. That sequence is necessary, but it is not sufficient.

The limiting factor soon becomes managerial rather than technical. Can operations trust evidence produced by the delivery path? Can a business owner accept smaller changes without demanding the certainty of a complete scope? Can governance distinguish a routine release from a consequential one? Can managers reward teams for service performance rather than for crossing project milestones?

If those answers remain no, the organisation may possess a continuous-delivery toolchain while continuing to behave like a conventional release factory.

The practical test is not whether software could be deployed at any moment. It is whether the organisation can make a safe, ordinary decision to deploy it without assembling a temporary institution around the act.

Continuous delivery becomes real when release ceases to be a drama. Until then, the automation is valuable, but the delivery remains continuous in name only.