VERIQTA PRESS

DEVOPS IN THE TRENCHES

OFFICIAL FREE CHAPTER PREVIEW

ANN OGECHI FELIX



TWO SELECTED CHAPTERS

Chapter 1 | Chapter 16

veriqtapress.com

COPYRIGHT & PERMITTED USE

DevOps in the Trenches
Copyright (c) 2026 Ann Ogechi Felix. All rights reserved.

Published by VERIQTA PRESS, an imprint of VERIQTA LIMITED.

This free preview is provided for personal reading and evaluation. You may share the official download link, but you may not reproduce, redistribute, modify, sell, or incorporate the contents into another publication without prior written permission from the copyright holder.

This preview contains selected chapters only and does not represent the complete publication.

Included chapters:
Chapter 1: Production Does Not Care About Your Transformation
Chapter 16: CI/CD Is a Production Service

Official website: https://veriqtapress.com/

CHAPTER 1

Production Does Not Care About Your Transformation

Distinguish transformation activity from production capability, identify gaps in ownership and recovery, and choose an improvement that can be tested.

Opening Scenario

The quarterly transformation review begins with a familiar slide. Thirty-four delivery pipelines have been standardized. Most application teams have moved their repositories. The cloud migration is ahead of schedule. Hundreds of employees have completed training. A new platform portal is live. Every program milestone is green.

Then a regional pricing service needs a routine configuration change.

The request enters a queue owned by release management. Security must approve a permission change, but the approver listed in the workflow left the company two months earlier. The application team can build the change but cannot reproduce the production configuration. The infrastructure team owns the cluster, the platform team owns the deployment templates, and a vendor owns the database extension. Nobody owns the complete release decision.

Three days pass before the change reaches a staging environment. The test data does not represent the production condition that caused the request. The deployment pipeline is automated, but the rollback requires a manual database step documented in a private team page. Operations asks for a release plan. The application team points to the pipeline. The platform

team points to the service owner. The service owner assumes operations will coordinate the recovery if anything fails.

The change is approved for a low-traffic period. It deploys successfully according to the pipeline. Twenty minutes later, customer requests begin timing out in one region. Infrastructure health remains green. Application error rates rise slowly rather than crossing the alert threshold. Customer support sees the problem first.

The incident channel fills with people, but authority remains unclear. One team wants to roll back. Another warns that rollback may corrupt recently written data. The vendor cannot join for forty minutes. An engineer who helped build the original service is contacted during personal time because the documented recovery path has never been tested.

Service returns after two hours. The configuration change was small. The failure was not.

At the review the next morning, every transformation milestone is still green. The pipeline completed its task. The cloud platform remained available. The change ticket contained the required approvals. Training completion did not decline. The program dashboard had no field for the missing service boundary, the untested rollback, the obsolete approver, the weak customer signal, or the dependence on one engineer's memory.

The organization had changed many visible things. It had not changed the conditions under which production work became safe.

Figure 1.1 follows the opening scenario from successful deployment to customer failure and recovery. The pipeline and infrastructure signals do not establish that the customer outcome is protected.

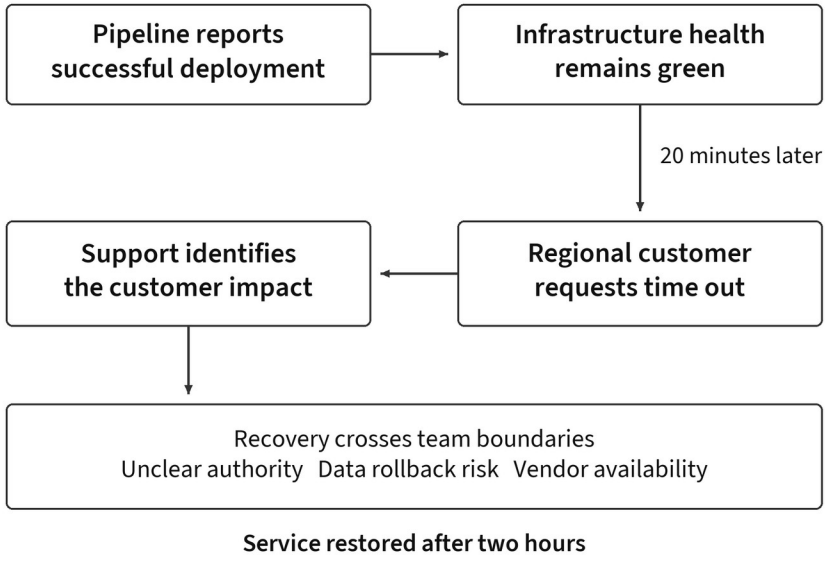


Figure 1.1. Deployment success does not establish customer success or recovery readiness. Scenario illustration: green pipeline and infrastructure signals coexist with customer timeouts; recovery still depends on authority, data safety, and specialist availability.

Transformation language becomes dangerous when it allows adoption to stand in for responsibility. Teams report movement while customers continue to experience the same delays, failures, and uncertain recovery. Automation has increased, but control has not.

Production does not evaluate the presentation. It responds to architecture, dependencies, permissions, queues, capacity, feedback, and decisions. When those elements disagree, the declared transformation has no special authority. The system behaves according to what was actually designed and maintained.

The first task is to determine which operating capabilities changed, what evidence demonstrates the change, and what production still exposes as unresolved.

The Myth of the DevOps Transformation

The transformation label suggests a controlled passage from an old state to a new one. It gives executives a program to fund, leaders a roadmap to report, and teams a destination to reach. Yet a temporary initiative cannot replace continuing responsibility for production.

A production system never reaches a final state. Applications, dependencies, threats, customer expectations, and operational knowledge all change. A delivery practice that worked for ten teams may become a constraint at fifty. A control designed for manual releases may create risk when inserted unchanged into an automated process. A platform can become another legacy dependency if its ownership and lifecycle are neglected.

For that reason, a credible transformation cannot mean that the organization has finished adopting DevOps. It must mean that specific operating capabilities have changed and that the organization can continue improving them. The evidence lives in work and outcomes, not in the name of the program.

Visible change and operating change

Visible change is easy to display. Teams move repositories, adopt infrastructure as code, form a platform group, migrate workloads, purchase observability products, or change job titles. These actions may be necessary. They are still inputs.

Operating change appears in the decisions surrounding production. A team can obtain an environment without waiting through an unmanaged queue. A change can move from commit to production through a path whose controls are known and testable. The people operating a service can see whether customers are receiving the intended outcome. A failed release has a bounded recovery path. Ownership remains clear when a dependency degrades, not only when everything is healthy.

An organization can complete the visible work while preserving the old system underneath it. A cloud migration may reproduce ticket queues and fixed-capacity assumptions. A continuous integration product may execute builds while integration remains infrequent and painful. A platform may publish templates while

application teams still need private intervention for every exception. The constraints survive beneath the new tools.

Transformation as identity

The label becomes more difficult to challenge when it turns into an organizational identity. Leaders have committed funding and credibility. Teams have reorganized. Vendors have been selected. Acknowledging that production capability has not improved can sound like rejecting the entire effort.

Adoption becomes a proxy for usefulness, completion for capability, and a program milestone for a customer outcome. Once a proxy enters executive reporting, teams learn to protect it. They count pipelines instead of examining release delay, count platform registrations instead of investigating bypasses, or count closed actions instead of verifying that a failure mode has been removed.

Ask what operating condition the initiative was supposed to change. Define the system boundary, identify the people affected, and establish a baseline that can guide decisions. Then examine whether capability improved without transferring unacceptable risk elsewhere.

Capability is contextual

DevOps Research and Assessment (DORA) studies the relationship between software delivery capabilities, delivery performance, and organizational outcomes. Its measurement guidance cautions against universal targets and comparisons between services with materially different contexts. Copying another organization's practices or thresholds does not establish that your own capability improved.[1]

A regulated payment system, an internal analytics tool, a mainframe settlement process, and a consumer mobile service face different consequences and constraints. Each can improve flow, feedback, stability, and recovery. The required capability must be appropriate to its context, observable, owned, and open to improvement.

---

1 Nicole Forsgren, Jez Humble, and Gene Kim, Accelerate (IT Revolution, 2018); Nathen Harvey, "DORA's Software Delivery Performance Metrics," DORA, accessed October 29, 2026, https://dora.dev/guides/dora-metrics/.

Transformation cannot be declared complete because a program delivered its outputs. Production capability must be demonstrated repeatedly, through routine work and the response to failure.

Why Tools Cannot Repair a Broken Operating Model

A tool executes the authority and assumptions given to it. It can make a decision repeatable, accelerate a workflow, preserve evidence, and reduce manual variation. It cannot decide who should own a service, resolve incentives that reward conflicting outcomes, or determine how much risk the organization should accept.

This limitation is easy to miss because tool adoption produces immediate evidence of movement. A pipeline runs. A portal creates an environment. A policy engine blocks a deployment. A dashboard shows a new metric. The interface is visible, while the operating model around it remains partly hidden.

Automation inherits the surrounding system

Consider a deployment pipeline introduced where development is measured on feature completion, operations on change avoidance, and security on issues blocked. The pipeline may remove manual steps, but those incentives remain. Their conflict moves into pipeline rules, exception queues, and approval logic.

The result may be faster execution of a slow decision path. It may also be a brittle path that works only for standard changes. When a change falls outside the template, teams return to private messages and manual intervention. The automation has not removed the old operating model. It has created a fast lane through one portion of it.

The same pattern appears in cloud adoption. Self-service infrastructure can reduce provisioning delay, but only if identity, network, security, cost, and lifecycle decisions have usable defaults and clear owners. Otherwise, a request moves quickly until it meets an unresolved boundary. Teams then accumulate standing

privileges, duplicate infrastructure, or build unofficial paths around the platform.

Tools amplify both strengths and weaknesses

Automation increases consistency when the underlying decision is sound. It also increases the speed and reach of a mistake. A manual misconfiguration may affect one environment. A shared template can reproduce the same error across hundreds. A centralized policy can prevent unsafe deployments, or it can stop urgent recovery work if exceptions were never designed. A common platform can reduce cognitive load, or it can create a dependency whose failure blocks every delivery team.

Before automating, ask which decision is being encoded, who owns it, what evidence it evaluates, and what happens when its assumptions fail.

Conditions for effective automation

First, the responsibility must be explicit. Someone must own the policy or workflow beyond its implementation. Ownership includes reviewing its continued validity, responding to failure, and deciding when an exception is justified.

Second, the normal path must be understood. Automating a process that nobody can explain often conceals rather than removes waste. Mapping the actual flow, including waits and rework, reveals whether the tool addresses the constraint or merely one visible step.

Third, degraded behavior must be designed. Identity services fail. Registries become unavailable, agents lose connectivity, and policy data becomes stale. The organization must decide whether work stops, continues with reduced assurance, or moves through a controlled emergency path.

Fourth, feedback must reach the owner. A platform team needs evidence about adoption, abandonment, exceptions, support demand, and downstream incidents. A security team needs evidence about false positives and bypass behavior. Without feedback, automation can remain technically healthy while becoming operationally harmful.

Tool selection still matters. Products differ in reliability, integration, security, maintainability, cost, and usability. Selection must follow operating design: a suitable tool will still expose unresolved decisions once it becomes central to production.

The correct sequence begins with the capability and its owner. Define the decision, evidence, constraints, failure behavior, and review cycle. Then select or build the mechanism that supports them. The tool becomes part of the operating model rather than a replacement for one.

The Difference Between Activity and Capability

Transformation programs produce activity because activity is controllable. Budgets can be allocated, training scheduled, repositories moved, pipelines created, controls configured, and migration waves completed. These outputs show that work occurred. They do not establish that the organization can perform the production task safely and repeatedly.

A capability is a dependable ability under stated conditions. It includes people, knowledge, authority, process, technology, evidence, and recovery. Deployment automation is an asset. The ability to release an important service on demand, observe customer effect, contain failure, and recover within an agreed boundary is a capability.

Program teams can report enabling work as if it were the outcome that work was meant to produce. Activity data arrives early and is easy to aggregate. Capability evidence spans systems and teams, and often demands more difficult interpretation.

Five tests of capability

The first test is repeatability. Can the organization perform the task without relying on a particular individual's memory or private access? A process that succeeds only when the most experienced engineer is present remains fragile, even if that engineer succeeds every time.

The second test is operating range. Under which conditions does the capability work? A deployment path may handle normal application changes but fail for database migrations, urgent

security fixes, regional recovery, or a degraded identity provider. Declaring the capability without its boundary hides the situations that matter most.

The third test is evidence. Can the responsible team show that the intended outcome occurred? A successful pipeline run proves execution, not customer success. A completed backup proves that data was written somewhere, not that the service can be restored. Evidence must match the claim.

The fourth test is ownership. Who maintains the capability, decides its standards, responds when it fails, and funds its improvement? Shared responsibility can be appropriate, but shared responsibility without decision rights usually becomes an escalation problem.

The fifth test is adaptation. Can feedback change the process, architecture, or policy? A capability that produces data but cannot act on it may become a reporting ritual. Learning requires authority and capacity, not only a retrospective meeting.

Measuring without creating theater

DORA groups its software delivery performance measures into throughput and instability. Use the measures to investigate outcomes for a service, with explicit definitions and context. Avoid converting them into universal targets or investing in perfect collection before beginning improvement.

These measures need not become a score. Their purpose is to expose constraints and support decisions. If deployment frequency rises while change failures and recovery remain stable or improve, the combined evidence may indicate greater capability. If frequency rises because controls moved outside the recorded path, the number may conceal risk.

Local success can also damage the wider system. A development team may reduce its cycle time by handing incomplete operational work to another group. A central platform may reduce infrastructure variation while increasing the queue for exceptions. A security team may close findings more quickly by rejecting risk rather than helping teams remediate it. Each group appears efficient. The end-to-end system becomes slower or less safe.

Capability must therefore be assessed at the boundary where value and consequence appear. Start with a service or value stream, not an organizational average. Follow work from request through production and recovery. Examine waits, rework, failure, customer effect, and ownership across team boundaries.

Activity deserves recognition when it builds the required capability. It should not be asked to prove more than it can. A migrated repository is a completed task. A reliable, traceable, recoverable change path is the result that matters.

Executive reporting should show whether the promised capability exists, where it fails, and which constraint requires a decision. Include the intended outcome, current evidence, material exceptions, risk transferred to other teams, and the next intervention. This requires more interpretation than counting completed migrations, but it reveals whether investment changed production behavior.

What Production Exposes About the Organization

Production is an integration test for the organization. It combines architecture, staffing, access, policy, vendor relationships, incentives, documentation, and time pressure. When a service fails or a change stalls, the visible technical symptom often reveals decisions made far from the component that broke.

This does not mean every incident is primarily organizational. Components have defects. Networks partition. Capacity is exhausted. Operators make mistakes. The important point is that the effect of those events depends on the surrounding system. A defect becomes a long outage when detection is weak, ownership is disputed, rollback is unsafe, or authority arrives late.

Queues reveal decision design

Long waits are often explained as insufficient automation or staffing. Sometimes that is correct. Sometimes the queue exists because the organization has not decided who can accept a risk. A change waits for several approvals because no single approver

owns the complete consequence. An environment request waits because network, identity, security, and cost controls were never converted into supported defaults.

Measure the wait and investigate the decision behind it. More processing capacity will not resolve uncertainty about who can accept a risk.

Incidents reveal ownership

Service catalogs and responsibility matrices describe ownership under normal conditions. Incidents test it. Who can declare severity, stop a rollout, communicate customer impact, involve a vendor, authorize degraded operation, or accept data-recovery consequences? If those decisions depend on hierarchy, personal relationships, or whoever responds first, the documented owner does not possess operational authority.

Repeated escalation to a small group also reveals where knowledge and trust have concentrated. The immediate response may require those people. The learning response should reduce the dependency through clearer service boundaries, access design, runbooks, rehearsal, and shared diagnosis.

Degradation reveals what the organization values

Systems can become slow, inconsistent, partially unreachable, stale, expensive, or difficult to recover before they become fully unavailable. The response to this degradation shows whether the organization has defined user-relevant behavior or only infrastructure health.

A dashboard may remain green because hosts are running while customers cannot complete a transaction. A team may meet an availability target while a critical minority of requests fail. The lesson is not that dashboards or objectives are useless. It is that evidence must represent the outcome the organization claims to protect.

Recovery reveals preparation

Recovery work exposes assumptions that normal operation can conceal. Documentation may describe commands without identifying decision points. Backups may exist without application

configuration, keys, dependency order, or validation data. Access may be available in the primary environment but unavailable when the identity path is impaired.

An untested recovery plan is a statement of intent. Demonstrated recovery requires rehearsal, observable success criteria, known dependencies, and an owner who can accept the recovered state. Detailed recovery engineering belongs later in this book, but the transformation lesson belongs here: readiness must be shown under representative conditions.

Production evidence without blame

Using production as evidence can become punitive if leaders treat every signal as a search for fault. That response teaches teams to protect metrics, narrow incident records, and avoid exposing weak boundaries. The organization then loses the information it needs to improve.

A useful review asks what information was available, which incentives and constraints shaped the action, and which controls failed to prevent, detect, or contain the event. It should also identify missing recovery dependencies. Preserve individual accountability while examining how the operating environment made the action possible or reasonable.

Transformation claims should become more specific after contact with production. The organization may discover that automation improved the normal path but not emergency change, that cloud adoption improved provisioning but not service ownership, or that observability improved detection but not decision authority. These findings do not automatically invalidate the investment. They identify the next constraint.

Follow production evidence across technical and managerial boundaries until the decisions shaping the outcome become visible.

Questions a Transformation Must Answer

A serious transformation begins with a production problem that matters, not a catalog of practices to adopt. It defines what must improve, for whom, under which constraints, and how the

organization will know. These questions create a basis for action without pretending that one model fits every service.

What outcome must change

Name the customer, user, operator, regulator, or business process affected. “Improve DevOps” is not an outcome. Reducing the time required to deliver a pricing change while preserving transaction correctness is an outcome. Restoring a critical service without relying on one person is an outcome. Producing traceable deployment evidence without a manual audit chase is an outcome.

Connect the outcome to a real consequence and state the guardrails. Faster change that increases unrecoverable data risk may fail the intended objective.

Where is the system boundary

Choose a service or value stream that can be examined end to end. Include the queues, control points, dependencies, and teams required to reach production and recover from failure. If the boundary ends at the application team, transferred work will disappear from the analysis.

The boundary may need to include a vendor, legacy platform, change authority, security control, or finance decision. Chapter 2 develops this mapping in depth. At this stage, the purpose is to prevent a local initiative from claiming a system outcome.

Who owns the result and the decisions inside it

Assign ownership at the level of consequence. Someone must be accountable for the service outcome. Different groups may own platform reliability, security policy, data integrity, or release control, but their decision rights and escalation paths must meet at the service boundary.

Ask who can stop a change, accept an exception, approve degraded operation, initiate recovery, and fund correction of a known risk. If the answer changes depending on who is available, the operating model is not ready for automation at scale.

What evidence will guide action

Select a small set of signals that reflect flow, stability, customer effect, and recovery. Use them to locate constraints and evaluate changes over time. Record definitions, scope, time windows, and limitations. Do not aggregate unlike services to create a reassuring enterprise number.

Combine quantitative and qualitative evidence. Queue time may show where work waits. Interviews and value-stream observation may explain why. Incident records may reveal recovery delay. Support contacts may show a customer problem that infrastructure metrics miss.

What failure behavior is acceptable

Every new path introduces failure modes. Decide what should happen when the pipeline, platform, identity provider, policy service, registry, or approval system is unavailable. Identify which operations may continue, which must stop, and who can authorize an exception. Test that decision before an urgent release requires it.

The goal is not uninterrupted operation at any cost. Some work should stop when assurance is unavailable. The important point is to make the condition deliberate rather than discover it during an incident.

What constraint should be addressed first

Transformation programs often launch several workstreams because every weakness appears important. That spreads attention and makes attribution difficult. Begin with the most consequential constraint that the organization can influence. Form a hypothesis, change the system, observe the effect, and retain or revise the intervention.

A focused intervention produces evidence that can guide the next decision. It also limits the risk of automating a process before the organization understands it.

How will learning change funding and priority

Continuous improvement fails when evidence has no route into planning. Teams may identify unsafe dependencies and recurring toil while investment remains tied to feature commitments. Leaders must reserve capacity and create a decision cadence that can act on operational evidence.

The review should ask whether the capability improved, what risk moved, what new constraint appeared, and whether the original outcome still matters. It should be willing to stop an initiative that produces adoption without value.

The result is a testable operating change with an owner, a boundary, evidence, and known failure conditions. That provides a sound basis for the first intervention.

War Story
The Platform That Passed Every Milestone

This story is a composite based on recurring enterprise patterns. Identifying details have been omitted.

A large organization approved an internal platform program after several years of slow releases, inconsistent environments, and difficult audit preparation. The business case was persuasive. Standard templates would reduce variation. Self-service would shorten provisioning. Central policy would improve security. Shared pipelines would create evidence automatically.

The platform team delivered on schedule. It built a service catalog, environment templates, continuous integration workflows, deployment automation, policy checks, and centralized logging. The first dashboard showed strong progress. Applications were registered. Repositories were connected. Engineers attended training. The program reported adoption by counting each completed migration.

Production behavior changed less than the dashboard suggested.

Most teams could create a development environment quickly, but production network rules still required a ticket reviewed by a small infrastructure group. Database changes remained outside the standard pipeline because ownership was split across application,

database, and release teams. Security policy blocked several legacy deployment patterns, but the exception process had no response target and no permanent owner.

Application teams learned where the platform ended. Some kept the standard pipeline for visible stages and performed sensitive steps manually. Others copied templates into their own repositories so they could modify them without waiting. A few services registered in the catalog but continued releasing through the previous system.

The portal remained available, pipeline jobs succeeded, and policy checks recorded decisions. Support demand was classified as onboarding assistance, so it did not challenge the reported success of the platform.

The failure chain

The immediate incident began with a routine certificate update for a revenue-critical service. The platform template supported automated rotation, but the service depended on an older component that loaded certificates only at startup. The application team had documented the exception in its own repository. The platform team did not track which services had modified the standard.

The pipeline replaced the certificate and reported success. One group of instances continued serving existing connections. Newly started instances failed to establish a downstream connection. Traffic shifted gradually, producing intermittent customer errors rather than a clean outage.

The observability platform showed healthy infrastructure and successful deployment status. Customer support reported transaction failures. The application team began diagnosis, then discovered that its usual rollback would restore the expired certificate. The security team rejected that option. The platform team could revert the template but could not decide whether the service should operate temporarily with the previous credential. The incident commander had technical coordination authority but no clear risk owner for the exception.

Recovery took longer than the certificate repair. Teams had to reconstruct the service's modified deployment path, obtain an

emergency security decision, restart instances in the correct order, and confirm that transactions had not been duplicated. The platform was not unavailable. It had done exactly what its shared workflow specified. The surrounding ownership model had failed to carry a known service exception into production control and recovery.

The first review focused on compliance with the platform. Leaders proposed mandatory adoption and tighter restrictions on template modification. That response would have increased visible conformity while preserving the missing decisions.

A second review followed the work across team boundaries. It found that the platform program had been measured on migration completion, not on reduction of release delay or recovery risk. The platform team owned components but not the service outcomes created through them. Application teams could modify templates without registering divergence. Security owned policy but not exception response. Service owners had responsibility without authority over several recovery decisions.

The correction

The organization stopped treating registration as adoption. It selected a small set of important services and mapped each delivery and recovery path. Platform teams recorded supported patterns and known divergence. Application teams retained responsibility for service behavior while platform owners accepted reliability obligations for shared paths. Security defined time-bounded exception authority and evidence requirements. Recovery exercises tested the interactions among those decisions.

Some platform features were simplified. Others gained stronger controls. A few services remained outside the standard path because forcing them into it would have increased risk. Their exceptions became visible, owned, and reviewed rather than hidden.

The dashboard began reporting support demand, completion through the supported path, unresolved divergence, change outcomes, and recovery exercises for selected services. These measures offered a more useful account of platform performance than the original adoption count.

The platform's technical foundation supported the correction. The mistake was assuming that delivering that foundation had already changed the organization's ability to operate production.

Lessons from the Trenches

- Transformation is a claim that requires production evidence. A program can deliver every planned artifact while the original constraint survives in approvals, ownership, recovery, or exception handling.
- Tools encode decisions. Before automating a path, identify who owns the decision, which evidence it uses, how exceptions work, and what happens when a dependency is degraded.
- Adoption is not the same as use, and use is not the same as outcome. Registration, training, migration, and pipeline execution are useful activity measures. They do not prove that customer impact, flow, stability, or recovery improved.
- The normal path is only part of the capability. A production practice must define its operating range, including database change, urgent correction, dependency failure, degraded identity, and recovery.
- Local efficiency can transfer risk. A team may shorten its work by creating a queue, exception, or operational burden elsewhere. Follow the work across the service boundary before declaring improvement.
- Production signals should lead to investigation rather than punishment. If teams fear that evidence will be used against them, metrics become defensive and weak boundaries remain hidden.
- Ownership requires authority. Naming a service owner is insufficient when another group controls rollback, access, exception approval, vendor escalation, or investment in known risk.
- Begin with one consequential constraint. Define the outcome, boundary, evidence, owner, and failure behavior. Improve it, observe the result, and then address the next constraint.

- The strongest transformation dashboard may contain fewer numbers. Each number should connect to a decision, a defined system boundary, and an outcome the organization is prepared to act upon.
- Exceptions are operating data. Repeated requests for manual access, template changes, urgent approvals, or support intervention show where the supported path does not match production reality. Hiding exceptions protects the program narrative and weakens the system.
- Recovery is where ownership becomes concrete. The person named as owner must be able to coordinate the technical, customer, security, data, and vendor decisions required to return the service safely.
- Improvement needs capacity. A retrospective can identify the right corrective work, but learning remains incomplete when planning and funding cannot make room for it.
- Context limits imitation. Practices from high-performing organizations can inform a decision, but their value depends on architecture, risk, regulation, staffing, and service purpose. Copy the reasoning before copying the shape.
- Transformation should reduce dependence on exceptional effort. If success still relies on late escalation, private knowledge, or repeated rescue by the same people, the operating capability remains immature regardless of the technology in place.

Operational Checklist

Use this checklist as an investigation, not as a maturity score. Apply it to one service or value stream. Record evidence, owners, and unresolved decisions.

Outcome and boundary

☐ Identify the customer, user, operator, or business process expected to benefit.

☐ State the production outcome the transformation is intended to change.

- ☐ Record the conditions that must not be sacrificed while pursuing that outcome.
- ☐ Map the path from requested change through production validation and recovery.
- ☐ Include external vendors, shared platforms, control functions, and legacy dependencies inside the boundary where they affect the outcome.

Ownership and authority

- ☐ Assign an accountable owner for the service outcome.
- ☐ Identify who can stop a release, approve an exception, authorize degraded operation, and initiate recovery.
- ☐ Verify that documented owners possess the access, funding path, and decision authority their responsibility requires.
- ☐ Record dependencies that rely on one individual's knowledge, relationship, or privilege.
- ☐ Define escalation paths for decisions that cross application, platform, security, operations, data, and vendor boundaries.

Capability and evidence

- ☐ Separate program activities from the operating capabilities they are meant to create.
- ☐ Define evidence for flow, stability, customer effect, and recovery at the service level.
- ☐ Record each measure's definition, scope, time window, and known limitation.
- ☐ Verify that a successful automated task demonstrates the claimed outcome rather than only tool execution.
- ☐ Examine waiting time, rework, manual intervention, bypasses, and exception queues.
- ☐ Compare current evidence with a useful baseline before declaring improvement.

Failure and recovery

- ☐ Identify how the delivery path behaves when identity, policy, registry, platform, network, or vendor services are degraded.
- ☐ Define which operations stop, continue, or require explicit exception authority.
- ☐ Test a representative rollback or recovery path, including data and dependency consequences.
- ☐ Verify that recovery does not depend on undocumented knowledge or unavailable credentials.
- ☐ Record how lessons from incidents, failed changes, and support demand alter the operating path.

Governance and learning

- ☐ Review whether team incentives reward conflicting outcomes or transfer work across boundaries.
- ☐ Assign a decision cadence for reviewing evidence and choosing the next constraint.
- ☐ Reserve capacity and funding for corrective work revealed by production evidence.
- ☐ Stop, narrow, or redesign initiatives that create adoption without improving the intended capability.

Further Reading

Beyer, Betsy, Chris Jones, Jennifer Petoff, and Niall Richard Murphy, eds. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016.

Read the principles and management sections for a production-centered treatment of risk, toil, automation, incidents, communication, and organizational engagement. Apply the ideas with attention to differences between Google's environment and your own.

DORA. *Accelerate State of DevOps Report 2024*. Google Cloud, 2024.

Use the report to examine the 2024 findings on user focus, stable priorities, platform engineering, leadership, and the trade-offs associated with emerging technology. Read the methodology and limitations before converting findings into policy.

DORA. "Capabilities Catalog." Accessed October 29, 2026. https://dora.dev/capabilities/.

Use this catalog to examine capabilities separately from products and isolated practices.

Forsgren, Nicole, Jez Humble, and Gene Kim. *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. IT Revolution, 2018.

Read this for the research model connecting software delivery capabilities, delivery performance, organizational outcomes, and culture. It also helps readers distinguish evidence from DevOps folklore.

Harvey, Nathen. "DORA's Software Delivery Performance Metrics." DORA. Accessed October 29, 2026. https://dora.dev/guides/dora-metrics/.

This guide explains the current five-metric model, the division between throughput and instability, service-level context, and common measurement failures. Its warnings about targets, comparisons, competition, and collection effort are as important as the metric definitions.

Kim, Gene, Jez Humble, Patrick Debois, John Willis, and Nicole Forsgren. *The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations*. 2nd ed. IT Revolution, 2021.

Read this for value-stream thinking, feedback, continual learning, and the operating practices that support DevOps beyond tool adoption.

Meadows, Donella H. *Thinking in Systems: A Primer*. Edited by Diana Wright. Chelsea Green Publishing, 2008.

Use this to understand feedback, delays, system boundaries, resilience, and why local interventions can produce unexpected behavior elsewhere.

Skelton, Matthew, and Manuel Pais. *Team Topologies: Organizing Business and Technology for Fast Flow of Value*. 2nd ed. IT Revolution, 2025.

Read this for team boundaries, cognitive load, interaction modes, platforms, and flow.

The Next Trench

Testing transformation claims against production exposes a boundary problem. A service depends on identity, networks, data, shared platforms, vendors, approvals, support, budgets, and operational knowledge as well as its repository and runtime.

When those relationships remain invisible, organizations improve components while risk accumulates between them. Documented ownership stops at a team boundary that production failure crosses.

Chapter 2 builds an enterprise production context map: the system that carries a change, serves a customer, and must recover under pressure. The map helps distinguish an improvement from a constraint that has merely moved elsewhere.

CHAPTER 16

CI/CD Is a Production Service

Opening Scenario

At 02:13, an authentication service began rejecting valid sessions. The incident team found the defect quickly: a configuration parser introduced in the evening release mishandled one legacy value. A corrected artifact was built, verified against the failing case, and ready for production before 02:40.

It did not reach production until 04:26.

The organization's delivery platform had failed during the same incident. Thousands of scheduled jobs had entered the queue after a shared template change caused retries across several repositories. Runner capacity was saturated. The orchestration service accepted new requests but reported no reliable estimate for when protected jobs would start. Canceling queued work created more API traffic and did not consistently release environment locks.

The incident commander asked for a manual deployment. Nobody could state what that meant. The documented emergency procedure assumed the pipeline control plane was available. The production credential used by the deployment adapter was intentionally inaccessible to engineers. A legacy script existed on an administrator's laptop, but it expected a mutable image tag and produced no durable record of who approved the action or which digest it deployed.

The team faced two different failures. The application defect was understood and reversible. The delivery failure was neither. Operators could not tell whether the platform was merely overloaded, whether a runner manager had failed, or whether the control plane had lost coordination with active workers. Several jobs displayed as running after their workers had disappeared.

Retrying the deployment risked two concurrent changes to the same environment.

The platform team eventually reserved a runner, cleared a stale environment lock, and promoted the corrected artifact through the normal deployment adapter. Service recovered. No secret was exposed and no conflicting deployment occurred, but the duration of the customer incident was determined less by the application fix than by the organization's inability to operate its delivery system under pressure.

The review rejected the easy conclusion that the company needed more runners. Capacity had contributed, but it was not the whole cause. The service had no owner for the complete path. Teams measured job duration but not queue age, lock age, or time from accepted change to verified production outcome. Shared and emergency work used the same scheduling class. Failure states were ambiguous. The fallback depended on a component that the fallback was meant to replace.

The corrective work began by naming CI/CD as a production service. The organization mapped its control plane and execution plane, assigned owners, established delivery service objectives, separated trust zones, introduced priority and fairness rules, instrumented queues and dependencies, and rehearsed a narrow emergency path using retained artifacts by digest.

A pipeline file describes desired work. It does not provide the service that makes the work dependable. That service has architecture, capacity, identities, failure modes, and recovery obligations. When it carries production authority, it deserves the same operational discipline as the systems it changes.

The incident did not prove that automation had failed as an idea. It proved that automation becomes infrastructure once people rely on it. Reliability then depends on the system around the script: scheduling, identity, isolation, state, evidence, and a tested way to operate when normal coordination is gone.

Pipeline Architecture, Ownership, and Service Levels

A delivery system is usually presented as a sequence of stages. That view is useful to a pipeline author and incomplete for an operator. The operating view begins with components, trust boundaries, dependencies, state, and ownership. It asks what happens after an event is accepted, where work waits, which identity authorizes each transition, and how the organization knows that an outcome is final.

Map the control plane and execution plane

The control plane receives repository events, evaluates configuration, schedules work, enforces approvals and policy, coordinates state, and records results. The execution plane runs jobs on hosted or self-managed workers. Between them sit artifact and cache stores, identity providers, secret brokers, deployment adapters, policy services, notification systems, and observability backends. Each is a dependency and often a separate failure domain.

Draw the path for one ordinary change and one emergency change. Record where state is authoritative, where credentials are issued, and where an action becomes externally consequential. A job marked complete by a worker may still require an artifact upload, policy decision, or deployment acknowledgment. If two components can both claim authority for the same environment, the architecture has a coordination problem, not merely a documentation gap.

Assign ownership at service boundaries

Application teams own the intent and correctness of their workflows. A delivery-platform team may own orchestration, shared templates, runners, integrations, and the service interface. Security may own control objectives and investigation support. Environment owners may control production admission. These responsibilities need an accountable owner for the end-to-end

path. Otherwise every component can meet its local target while the user waits for a change that never completes.

Define support hours, on-call coverage, escalation, change authority, and dependency ownership. A hosted provider does not remove the service owner. It changes which failures can be repaired directly and increases the importance of provider status, exportable evidence, contractual commitments, and alternate operation.

Classify delivery paths by consequence

Not all pipelines deserve identical objectives. A documentation preview, a routine development build, a payment-service release, and an emergency security fix have different consequences. Classify paths by business impact, production privilege, reversibility, and time sensitivity. The classification should affect capacity, isolation, support, recovery, and change control, not merely dashboard color.

Avoid declaring the whole platform critical and then providing no distinction inside it. Reserve the strongest guarantees for work whose delay or corruption creates material harm. Lower-consequence work still needs predictable service, but it may accept longer queues or reduced support outside business hours.

Define service levels around the user journey

Useful indicators include event acceptance, queue wait, time to start, time to a conclusive result, successful publication or promotion, and time to recover the delivery service. Measure by workload class and dependency. A global success rate can look healthy while one protected environment is blocked for hours. A job failure caused by a test is not the same as platform failure, even though both produce a red pipeline.

Objectives should express a real operating promise: for example, a defined proportion of protected release jobs start within an agreed interval when their prerequisites are healthy. Pair the objective with exclusions and dependency assumptions, but do not use exclusions to hide chronic integration failure. Include emergency capability, evidence retention, and configuration-propagation expectations where they matter.

Manage the service as a product

Publish supported workload classes, limits, maintenance policy, deprecation windows, and escalation routes. Review demand, recurring failure, manual intervention, and user friction. A successful platform is not one that centralizes every decision. It provides dependable paths, makes constraints visible, and gives teams a controlled way to handle exceptional needs.

Architecture and ownership establish who can act. The next question is where untrusted pipeline code is allowed to act. That boundary is the runner fleet.

Review hidden dependencies before they become outages

Record provider quotas, webhook delivery assumptions, certificate and token expiry, DNS, time synchronization, artifact retention, and identity availability. These dependencies are easy to omit because no pipeline stage names them. Test what users observe when each one slows or fails. A dependency that can stop every protected release belongs in the service risk register even when another team or vendor operates it.

Use architecture reviews to remove accidental single points of coordination, not to demand redundant copies of everything. Some components can fail closed and wait. Others need fault-tolerant operation because they govern emergency change. The classification, evidence, and recovery objective should determine where redundancy is justified.

Runner Isolation and Workload Boundaries

A runner executes repository-controlled code. That code may be trusted release automation, an experimental branch, or a pull request from an unknown contributor. Treating all three as ordinary compute collapses the boundary between untrusted input and production authority.

Choose the runner model deliberately

Hosted runners reduce infrastructure ownership and can provide a clean environment per job. Self-managed runners offer network

proximity, custom hardware, and tighter placement control, but they add patching, image maintenance, credential protection, capacity, and forensic responsibility. Persistent runners are efficient for stable internal workloads, while ephemeral runners reduce residual state. None of these models is safe by label. The configuration and trust zone determine exposure.

Inventory runner groups by who can submit work, which repositories can use them, what identities they can obtain, which networks they can reach, and what state survives a job. A runner that can be selected through a broad repository label is not dedicated merely because its virtual machine has a special name.

Separate workload classes

Untrusted contributions should not share execution state, caches, credentials, or network access with protected releases. Separate ordinary validation, privileged build or signing work, deployment activity, and administrative automation. Use distinct runner groups, accounts, subnets, identities, and policy where consequence requires it. A logical label alone is weak isolation if every group ultimately reaches the same host daemon or cloud role.

Fork and pull-request workflows need special care. Configuration from the proposed change may execute before maintainers approve it. Do not expose protected secrets merely because the target repository is trusted. Review the event semantics of the chosen platform, especially workflows that run with the target repository's authority while evaluating untrusted content.

Remove ambient privilege

Issue short-lived job identity scoped to the repository, environment, action, and duration. Prefer workload federation or a secret broker over long-lived cloud keys. Restrict outbound network access to required services. Block access to host sockets, cloud metadata, unrelated workspaces, and control-plane credentials. Privileged containers and mounted container-engine sockets should be treated as host-equivalent access.

Credentials must not persist in workspace files, process arguments, debug archives, or reusable images. Redaction helps, but it cannot protect a secret deliberately transformed or exfiltrated by hostile code. The primary defense is to prevent an untrusted job from receiving the secret.

Treat caches and workspaces as cross-job channels

A cache can carry executable content from one job to another. Restrict write authority, separate protected and untrusted scopes, validate content where possible, and include relevant identity in keys. Persistent workspaces require reliable cleanup and detection of residue. Ephemeral workers reduce this burden but do not correct an over-broad identity or network policy.

Operate the fleet

Version runner images and agents, patch them through controlled rollout, record which image executed each job, and retire unsupported versions. Monitor unexpected registrations, idle but privileged workers, image drift, failed cleanup, unusual network destinations, and jobs that escape normal duration. Maintain enough trusted capacity for protected work without forcing it onto a weaker pool during demand spikes.

Isolation introduces cost and cold-start latency. Apply it according to consequence, then make the trade-off explicit. A shared fleet can be appropriate for low-risk work. A production deployment should not inherit that risk accidentally because the queue was shorter.

Verify the boundary continuously

Configuration reviews are not enough. Submit controlled canary jobs that attempt prohibited network access, metadata access, cross-workspace reads, and unauthorized credential requests. Confirm that the platform blocks the action and produces evidence. Test cleanup by planting harmless markers and checking that later jobs cannot recover them.

When a boundary fails, identify every workload that used the affected pool during the exposure window. Preserve runner images, registration history, job identity, network records, and

artifact publication. Rotate or revoke credentials that could have been reached. Rebuilding the worker without assessing artifacts and downstream actions restores capacity but not confidence.

Document who may place a repository into each runner class and review that entitlement periodically. A well-isolated privileged pool still fails if broad groups can route arbitrary workflows into it.

Failure Handling, Concurrency, and Queue Pressure

Delivery failures are rarely limited to a clean pass or fail. A worker can lose contact after performing an external action. A deployment adapter can time out while the target continues. A cancellation can arrive after a job enters its critical section. Reliable operation depends on defining these ambiguous states before retries and concurrency turn them into duplicate change.

Distinguish failure domains and outcomes

Classify failures across event ingestion, configuration evaluation, scheduling, runner provisioning, execution, artifact transfer, policy, target integration, and result recording. Preserve the difference between a workload failure and a platform failure. More importantly, distinguish not performed, performed, failed, and outcome unknown. An unknown deployment outcome is not safe to retry until the target state is reconciled.

Design consequential steps to be idempotent where possible. Use stable operation identifiers, target-state checks, and reconciliation. Database migrations, ticket creation, notifications, and external provisioning often need explicit deduplication or compensation. A retry policy should name the errors it handles, its limit and backoff, and the evidence required before another attempt.

Control concurrency at the right scope

Concurrency may be safe across independent tests and unsafe across changes to one environment. Define groups around the resource whose state must remain coherent: an environment,

tenant, database, shared account, or release train. Decide whether a newer run should wait, cancel an older queued run, or supersede it. Never assume cancellation instantly stops external work.

Prevent stale work from deploying after newer work. Before a consequential action, recheck revision, approval, artifact identity, and environment state. Environment locks need owners, leases, expiry, and recovery procedures. A lock with no safe break mechanism converts a failed coordinator into an indefinite outage.

Treat the queue as operational state

Queue depth alone is weak. Measure age, start latency, workload class, blocked dependency, cancellation age, and oldest protected job. Bursts may come from a monorepo fan-out, scheduled work, mass retry, provider recovery, or an incident. Each needs a different response. Autoscaling cannot help when a downstream rate limit or serialized environment is the bottleneck.

Define priority without permitting permanent starvation. Reserve capacity for emergency and protected work, cap noisy tenants, and apply fair sharing across teams. Admission control may reject or defer low-value duplicate work before it consumes scarce execution. Coalesce superseded validation when the organization does not need every intermediate revision.

Plan capacity and backpressure

Model normal demand, burst demand, runner provisioning time, job duration, external quotas, and cost limits. Warm capacity reduces start latency but costs money. Ephemeral capacity improves isolation but may start slowly. Use evidence to choose the balance for each class. Protect the control plane from retry storms with bounded client retries, jitter, circuit breaking, and rate-aware scheduling.

Recover the queue deliberately

After an outage, replaying every accepted event may recreate obsolete work and overload dependencies. Reconcile desired state first. Cancel superseded runs, retain required audit evidence, prioritize current protected changes, and increase capacity only

where downstream systems can absorb it. Communicate which work will not be replayed.

A healthy queue is not an empty queue at all times. It is a governed commitment: work is admitted consciously, ordered according to consequence, and given a conclusive outcome without compromising shared state.

Make failure policy visible to pipeline authors

Platform defaults should expose timeout, retry, cancellation, and concurrency behavior in plain language. Silent defaults become production behavior even when authors never chose them. Provide safe patterns for common operations and require explicit review for unbounded fan-out, privileged retries, or jobs that can outlive their initiating run.

Review queue incidents as system events. Ask which work was admitted, which dependency saturated first, why clients retried, whether priority behaved as intended, and how long the oldest consequential work waited. Adding workers is only a complete correction when execution capacity was the demonstrated bottleneck.

Publish queue status and degraded-service decisions to users. When teams cannot see whether work is delayed, rejected, or unsafe to retry, they create parallel runs and worsen the condition.

Observability for Delivery Systems

A delivery platform can report green infrastructure while its users wait, retry, and bypass. Observability must follow the delivery journey from accepted event to verified outcome and make platform failure distinguishable from the software feedback the platform is meant to produce.

Start with the user journey

Give each delivery attempt a correlation identity that connects the source event, evaluated workflow, scheduled jobs, runner instance, artifact digest, policy decision, environment action, and final verification. Preserve parent-child relationships for fan-out

and fan-in. Without this context, operators assemble an incident from timestamps across unrelated consoles.

Define completion from the user's perspective. A job finishing on a runner is not complete if its artifact is still uploading or its deployment acknowledgment is missing. Record when the platform accepts responsibility, when work becomes runnable, when it starts, when an external side effect occurs, and when the outcome becomes conclusive.

Measure demand, latency, errors, and saturation

Measure event rate, queued work, queue age, start latency, execution duration, end-to-end completion, runner utilization, provisioning time, control-plane limits, dependency latency, retries, cancellations, stale runs, manual interventions, and environment-lock age. Segment by workflow class, tenant, runner group, environment, and failure domain. Avoid high-cardinality labels that expose secrets or make telemetry unusable.

A pipeline success rate mixes platform health with expected test and policy failures. Build a taxonomy that attributes failure to workload, platform, dependency, policy, or unknown cause. Track unknowns because they reveal missing evidence. Measure repeated retries and manual restarts as service friction even when the final run succeeds.

Make logs and traces safe enough to use

Use structured logs with stable event names and identifiers. Record decisions, not only stack traces: why a job waited, which concurrency rule applied, which runner image executed it, and why a credential request was denied. Redact secrets before ingestion and restrict access to logs that may contain source, environment names, or deployment details.

Trace calls across webhooks, schedulers, runner managers, artifact services, policy engines, and deployment adapters where the platform allows it. Sampling must preserve rare failures and protected delivery paths. Audit records for workflow changes, approvals, administrative actions, runner registration, and emergency use require stronger retention and integrity than ordinary debug telemetry.

Alert on user harm and loss of control

Alert on sustained start-latency breaches, old protected work, runner-pool exhaustion, widespread unknown outcomes, stuck locks, failed credential issuance, artifact publication failure, and loss of audit evidence. Do not page the platform team for every application test failure. Route actionable product feedback to the owning team and platform symptoms to platform responders.

Use service objectives to drive improvement

Review objectives by workload class and inspect the slowest and most retried journeys. Tie improvement to causes such as insufficient warm capacity, noisy-neighbor behavior, provider limits, template regressions, or approval latency. A dashboard should support a decision: add capacity, isolate a tenant, repair an integration, change a limit, or communicate degraded service.

Observability should also survive the primary platform. Export essential audit and operational evidence to a protected destination so a control-plane outage does not erase the data needed to understand it. That evidence becomes critical when normal delivery is unavailable.

Build operator views around questions

An incident view should answer whether events are being accepted, where work is waiting, which pools are saturated, which dependencies are failing, and whether any environment has an unknown outcome. A service-owner view should show objective performance, demand growth, top causes of delay, tenant imbalance, and recurring manual work. A team view should explain why its run is waiting or failed without requiring platform access.

Validate telemetry during failure. Disconnect a runner manager, exhaust a quota, delay an artifact service, and create a stale lock. Confirm that the signals identify the failing boundary and that alerts reach the owner able to act. Telemetry that only works during successful execution is documentation, not operational evidence.

Retain enough history to compare incidents with normal demand and recent platform changes. Without a baseline,

operators may mistake a predictable daily burst for an anomaly or overlook a slow degradation hidden by weekly averages.

Recovering When the Deployment System Is Unavailable

The delivery system is most valuable when production needs change urgently. If its failure removes the only safe way to deploy, a minor defect can become a prolonged incident. Recovery planning must preserve control while restoring the minimum capability needed to change production.

Define the minimum trusted delivery capability

For many services, the minimum path must retrieve an already trusted artifact by digest, authenticate with short-lived authority, target a defined environment, execute a constrained change, record approval and operator identity, observe the result, and reverse or stop if conditions worsen. It may not need the complete development pipeline. It does need an unambiguous artifact and a durable record.

Identify which parts must remain independent of the primary platform. An emergency procedure that requires the failed scheduler, secret broker, identity provider, or artifact store has not removed that dependency. Complete independence is expensive, so choose it according to consequence and document shared failure modes honestly.

Separate outage from compromise

A capacity failure, provider outage, configuration error, and suspected compromise require different responses. Failover may help an outage and replicate a compromise. If integrity is in doubt, freeze normal promotion, preserve evidence, revoke affected identities, identify artifacts produced during the exposure window, and use a trusted alternate only after its inputs and authority are established.

Prepare a narrow emergency path

Preauthorize roles and decision rights. Require two-person control where consequence justifies it. Limit environments, operations, artifact sources, and time. Generate tamper-resistant records and expire emergency access automatically. The path should be easier to audit than an improvised administrator shell, even if it is less feature-rich than normal delivery.

Do not store an unrestricted production key in a sealed document and call that recovery. Protect credentials through an independent broker or controlled break-glass mechanism, test issuance, and monitor every use. If manual commands are unavoidable, version and review the procedure, validate target state before action, and capture exact commands and results.

Back up authoritative state

Protect workflow definitions, shared templates, policy, runner configuration, integration settings, environment mappings, identity configuration, audit records, and references to secrets. Preserve immutable artifacts and provenance for the supported rollback period. Caches and disposable workers can usually be reconstructed; authoritative configuration and evidence cannot.

Test restoration into an isolated environment. A configuration export that cannot be imported under current product versions is not a recovery capability. Record recovery dependencies, expected time, data loss, and who can authorize return to service.

Reconcile after restoration

When the primary service returns, do not simply resume every queued job. Reconcile environment state, artifact digests, approvals, locks, and actions taken through the alternate path. Cancel stale runs and restore ordinary authority. Review and close emergency access. Recreate missing evidence and record any gap that cannot be repaired.

Exercise under realistic conditions

Run drills that remove a real dependency and require responders to deliver a low-risk change through the alternate path. Test both

outage and suspected-compromise scenarios. Measure time to decide, time to obtain authority, time to deploy, evidence completeness, and reconciliation effort. A procedure read aloud in a meeting is not a recovery test.

The result is not a second full platform maintained for appearance. It is a deliberately small, trustworthy capability matched to production consequence. Once the service can deliver dependably, the next trench is deciding which risks the delivery system must test before a change deserves release.

Govern return to normal operation

Name the authority that declares the primary service trustworthy again. Restoration of availability is insufficient after suspected compromise. Verify administrative configuration, runner registration, integration credentials, policy state, audit continuity, and artifacts produced during the affected period. Record which evidence supports the decision and which uncertainty remains.

After every use of the alternate path, review whether it was narrow enough and whether responders could use it without private knowledge. Correct the normal platform as well as the fallback. An emergency path that is easier than ordinary delivery will attract routine use; one that is impossible under pressure will be abandoned. The design must sit between those failures.

War Story The Pipeline Outage That Extended a Minor Incident

This is a composite scenario built from recurring patterns in enterprise delivery incidents. Names, timings, technologies, and organizational details are fictionalized. It is not presented as a verified account of one company.

A financial-services company used one shared delivery platform for more than four hundred repositories. The platform team had standardized templates, centralized production credentials, and removed direct administrator access from most application teams. Delivery was faster and routine changes were easier to audit.

On a Monday morning, a malformed response from an external identity service caused one customer-facing API to reject a small class of requests. Engineers prepared a configuration correction within twenty minutes. The artifact already existed in the trusted registry. Only promotion to production remained.

The second failure

The production pipeline stayed queued. A shared validation template released earlier that morning had triggered duplicate workflows across large monorepositories. Developers retried slow jobs, which added demand. Runner autoscaling reached its cloud quota, while the scheduler continued to accept work. A policy integration also slowed because every queued run requested the same authorization check.

Operators first treated the problem as insufficient compute. They raised a concurrency limit, but the identity service and artifact proxy became the next bottlenecks. Start latency increased. Several workers lost contact after receiving jobs, leaving the control plane uncertain whether consequential steps had begun.

The emergency path that was not independent

The incident commander invoked the documented emergency process. It used a manually approved pipeline on a reserved runner. The runner was reserved, but approval, credential issuance, artifact lookup, and environment locking still depended on the same control plane. The process waited behind the failure it was meant to bypass.

A senior administrator proposed deploying from a workstation. The production cluster no longer accepted personal credentials, which was an intentional control. Another engineer found an older automation script, but it selected artifacts by release tag and could not produce the organization’s required deployment record. Using it would restore speed by discarding the controls introduced after an earlier audit.

Regaining control

The platform team stopped admission of low-priority scheduled work and disabled automatic retries in the affected template. It

isolated the oldest protected jobs, verified that no deployment had reached the target, broke a stale environment lock under two-person approval, and assigned one known-clean runner to the correction. The artifact was promoted by digest and the API recovered.

The customer defect lasted less than an hour in engineering terms. Customer impact lasted nearly three hours because the organization could create a fix but could not reliably move it. The delivery platform had become part of the production critical path without receiving production-class capacity, failure semantics, or recovery design.

What changed

The company mapped the full delivery journey and named one team accountable for it. Workload classes received explicit priority and fairness rules. Emergency capacity was reserved end to end, including identity, artifact, and environment-control dependencies. Retries became bounded and jittered. Environment locks gained leases, owners, and a controlled break procedure.

Runner pools were separated by trust and consequence. Untrusted validation could no longer consume protected deployment capacity. Dashboards added queue age, oldest protected job, unknown outcomes, lock age, runner provisioning time, and dependency saturation. The team defined objectives for start latency and conclusive completion rather than celebrating average job duration.

The alternate path was rebuilt around a narrow capability: retrieve a previously trusted artifact by digest, acquire short-lived emergency authority, deploy to a named environment, record the decision, and observe the result. It did not compile code or accept arbitrary scripts. Quarterly drills tested both platform outage and suspected compromise, followed by reconciliation into the restored system.

The management lesson

Leadership initially asked why an expensive platform needed reserved idle capacity. The incident reframed the cost. The relevant comparison was not utilization of a runner fleet. It was the

business cost of being unable to change production safely during an incident. The company paid for limited warm capacity, independent evidence, and recovery drills because those controls shortened the period in which it could not act.

Centralization had not been the mistake. Centralization had made the platform valuable and increased its consequence. The mistake was operating that authority as a collection of templates rather than as a production service.

Lessons from the Trenches

Pipeline configuration is not the platform

- The service includes event ingestion, orchestration, execution, identities, artifacts, policy, environments, state, telemetry, and dependencies. Operate the complete path.

Ownership must follow the user journey

- Component owners are necessary. One accountable service owner must still explain why accepted work did not reach a conclusive outcome.

Runners are a security and reliability boundary

- They execute repository code and may reach sensitive networks and identities. Separate workload classes, remove ambient privilege, and treat persistent state as a cross-job channel.

Unknown is an operational state

- After a timeout, a consequential action may have succeeded. Reconcile target state before retrying. Design operations for idempotence, deduplication, or compensation.

Queues express policy

- Priority, fairness, admission, cancellation, and reserved capacity determine whose work proceeds under pressure. Make those decisions explicit before an incident.

Observe waiting and intervention

- Average duration hides old protected jobs, repeated retries, stale locks, and manual recovery. Follow the path from accepted event to verified outcome.

Emergency delivery must be smaller, not uncontrolled

- Limit its artifacts, operations, environments, identities, duration, and authority. Record every use and reconcile when normal service returns.

Outage and compromise are different

- Failover can restore an unavailable service and spread a corrupted one. Decide which condition exists before moving authority.

Recovery depends on retained artifacts and evidence

- A crisis is not the time to rebuild a prior release or rediscover which digest ran. Preserve artifacts, provenance, configuration, and audit records for the required recovery period.

The practical review

- Choose one protected deployment. Trace its identities, queues, runner, artifact, policy, environment lock, external dependencies, telemetry, and fallback. Then remove the primary control plane in a drill. Every assumption that cannot be demonstrated becomes part of the operating backlog.

Capacity is part of control

- When protected work has no assured capacity, incident pressure pushes teams toward weaker runners, shared credentials, and manual deployment. Reserve capacity according to consequence and test the downstream services needed to use it.

A retry is a change decision

- Retrying is safe only when the previous outcome is known or the operation is designed for repetition. Treat automatic retries of consequential steps as reviewed production behavior, not harmless resilience.

Evidence must outlive the worker

- Ephemeral execution improves isolation, but investigation still needs runner identity, image, inputs, logs, network decisions, artifact digest, and external outcomes. Export that evidence before the worker disappears.

Standardization increases responsibility

- A shared platform can reduce variation and improve control. It also concentrates outage and compromise consequence. The more teams depend on one delivery service, the stronger its ownership, change discipline, capacity management, and recovery obligations become.

Operational Checklist

Architecture, ownership, and service levels

☐ A named owner is accountable for the end-to-end CI/CD service, including hosted-provider dependencies.

☐ The control plane, execution plane, state stores, identity paths, artifact services, policy services, and deployment adapters are mapped.

☐ Workload classes are defined by production consequence, privilege, reversibility, and time sensitivity.

☐ Service indicators distinguish event acceptance, queue wait, start latency, execution, publication, promotion, and verified completion.

☐ Service objectives are segmented by workload class and include a tested emergency-delivery expectation where required.

☐ Support hours, on-call ownership, escalation, maintenance, and dependency contacts are published.

Runner and workload boundaries

☐ Runner groups record who can submit work, what credentials they can obtain, which networks they can reach, and what state persists.

- ☐ Untrusted contributions cannot access protected secrets, caches, networks, runners, or production identities.
- ☐ Privileged jobs use short-lived, narrowly scoped workload identity rather than shared long-lived credentials.
- ☐ Runner workloads cannot reach host sockets, cloud metadata, unrelated workspaces, or control-plane credentials without an approved exception.
- ☐ Runner images and agents are versioned, patched, attributable to jobs, and retired through controlled rollout.
- ☐ Cache write authority and namespaces follow trust boundaries and prevent untrusted-to-protected reuse.

Failure, concurrency, and queue pressure

- ☐ Consequential operations define idempotence, deduplication, compensation, or reconciliation before retry.
- ☐ The platform distinguishes failed, cancelled, not performed, performed, and outcome unknown.
- ☐ Concurrency groups and environment locks match the resource whose state must remain coherent.
- ☐ Locks have owners, leases, age telemetry, and an authorized recovery procedure.
- ☐ Stale or superseded runs are prevented from applying older intent to a newer environment.
- ☐ Queue policy defines admission, priority, fairness, tenant limits, emergency reservation, and replay after outage.
- ☐ Capacity planning includes provisioning delay, downstream quotas, burst demand, cost limits, and noisy-neighbor behavior.
- ☐ Retries are bounded, back off with jitter, and cannot amplify a provider or control-plane failure.

Observability and evidence

- ☐ A correlation identity connects source event, workflow, jobs, runner, artifact digest, policy, deployment, and verification.

- ☐ Dashboards expose queue age, oldest protected work, runner saturation, unknown outcomes, stale locks, retries, and manual intervention.
- ☐ Alerts target user harm, loss of control, and evidence failure rather than every application test failure.
- ☐ Administrative, approval, runner-registration, and emergency-use records are retained outside the primary failure domain where required.

Recovery and emergency delivery

- ☐ The minimum trusted delivery capability is defined for critical services and retrieves artifacts by immutable digest.
- ☐ Emergency authority is narrow, short-lived, monitored, tested, and followed by reconciliation and access closure.
- ☐ Outage and suspected-compromise procedures differ, including freeze, revocation, evidence preservation, and alternate-builder trust.
- ☐ Workflow, template, policy, runner, identity, environment, audit, artifact, and provenance recovery has been restored in a drill.
- ☐ A realistic exercise has demonstrated delivery while the primary platform or a named dependency is unavailable.

Further Reading

NIST SP 800-204D, Strategies for the Integration of Software Supply Chain Security in DevSecOps CI/CD Pipelines

A concise architecture and control reference for integrating supply-chain security into CI/CD. Read it for trust boundaries, pipeline stages, and the relationship between security services and delivery workflows.

CISA and NSA, Defending Continuous Integration/Continuous Delivery Environments

Operational guidance on protecting identities, secrets, runners, repositories, artifacts, and administrative access in CI/CD environments. Use it to challenge assumptions about where compromise can propagate.

OWASP, CI/CD Security Cheat Sheet

A practical threat-oriented review of pipeline configuration, dependencies, credentials, workers, and third-party services. It is useful as a review companion, not a substitute for an architecture model.

Google, Site Reliability Engineering: Release Engineering

Explains reproducible, automated, hermetic, and policy-driven release practices at scale. It helps connect delivery mechanisms to operational reliability and organizational control.

Google, The Site Reliability Workbook: Implementing SLOs

A practical guide to selecting service indicators and objectives. Apply its reasoning to developer-facing delivery journeys rather than copying availability targets from customer-facing services.

GitHub Docs, Security hardening for GitHub Actions

Official mechanism-level guidance for event semantics, tokens, third-party actions, and runner risk. Review the current documentation against the specific events and runner models your organization uses.

GitLab Docs, Runner security and runner autoscaling

Official operational detail on executor trust, self-managed runner risks, concurrency, autoscaling, and fault-tolerant runner management. Use it to inspect concrete configuration trade-offs.

OpenTelemetry Documentation

The vendor-neutral model for traces, metrics, logs, context propagation, and semantic conventions. It supports end-to-end correlation across delivery components, subject to what each platform exposes.

Jez Humble and David Farley, Continuous Delivery

The foundational treatment of deployment pipelines, feedback, repeatability, and low-risk releases. Some product examples have aged, but the operating principles remain valuable.

The Next Trench

A dependable delivery service can start work predictably, isolate it, govern shared capacity, explain failure, and recover when its normal path is unavailable. That capability answers whether the organization can move change. It does not answer whether the change has confronted the risks that matter.

A long pipeline can still create false confidence. Thousands of fast checks may miss contract drift, production-scale contention, dependency failure, policy changes, corrupted data assumptions, or behavior that appears only under real traffic. A staging environment can resemble production and still omit the interaction that causes harm.

Chapter 17 moves from delivery mechanics to evidence. It asks teams to choose tests by failure consequence, combine contract, integration, performance, and resilience testing, manage data and environment fidelity, verify infrastructure and policy changes, and test safely in production where preproduction cannot reproduce the condition.

The next question is not how many tests ran. It is which operational claim each test supports, which risk remains untested, and how the organization will learn before exposure becomes an incident.

VERIQTA PRESS

CONTINUE THE JOURNEY

You have reached the end of this free preview.

Explore the complete publication at

veriqtapress.com

Copyright (c) 2026 Ann Ogechi Felix