# Speed Versus Code Quality

## Idea In Short

Treat speed and quality as correlated outcomes, not a trade. Tier every system by two questions: what a defect would cost and whether the decision can be reversed. Hold authentication, data pipelines and core domain logic to strict standards, and let prototypes and internal tools flex. Record each deliberate shortcut as debt with an owner and a repayment date, and place it in sprint planning. Fix slow builds and flaky tests first, because they drain velocity quietly. In regulated domains, automate compliance evidence inside the pipeline. Track the DevOps Research and Assessment [DORA] metrics alongside leading indicators, and review the balance every quarter.

Teams that deploy most often also tend to fail least often. That is the pattern in the research of the DevOps Research and Assessment [DORA] program, yet speed and quality still get framed as a trade during planning meetings. Product leaders push for more features while defect counts and on-call load keep climbing. This article explains how engineering teams balance the two in practice: where rigor is required, which practices return the investment and how to measure the outcome.

## Why speed versus quality is usually the wrong framing

Cutting corners pays for output in a single sprint. It rarely does over two quarters, because the time saved returns as rework, firefighting and modules that nobody wants to touch. Deployment frequency and change failure rate also tend to move together among DORA's top performers, so speed and stability correlate more often than they compete.

Startups know the pattern well. A team skips integration tests to meet a launch date, ships on time and then spends two quarters stabilizing a checkout flow that fails under real traffic. The start was quick, but the roadmap that followed was not.

The real contest is between intended and unintended shortcuts. A deliberate shortcut

carries a known cost and a repayment plan, while an accidental one gets discovered at 2 am. Martin Fowler's technical debt quadrant sorts debt along two axes: deliberate or inadvertent, and prudent or reckless. A prudent, deliberate shortcut ships sooner with the cost understood, and Fowler notes that such debt may not be worth paying down when the interest payments are small. A reckless one assumes the team cannot afford the time to write clean code. The inadvertent form appears when even strong teams learn what the design should have been only after they build it.1

## Deciding where quality is non-negotiable and where it can flex

A prototype that validates demand does not need a payment flow. Internal tools and experiment code can tolerate bugs that would never be allowed in authentication, data pipelines or core domain logic. Two questions resolve most cases: what would a defect cost here, and is the decision reversible? A two-way door sits behind a feature flag, while a schema migration on production data normally does not.

A simple way to apply the rule is to tag each service as strict, standard or flexible and to publish the tags where everyone can see them. The tag then sets the review depth, the test expectations and the release process for that service.

Debt incurred for a specific reason should work like a loan. Record it, assign an owner and set a repayment date, for example before the service handles 10,000 transactions per day. Put these entries in sprint planning rather than in pull request comments, so that product and engineering can both see them. In regulated domains, the flexible zone shrinks to a small size.

## Engineering practices that protect speed over time

Velocity leaks first in the pipeline. A build that fails at random and takes 40 minutes teaches engineers to batch work and rerun jobs, which is the opposite of what fast teams want. Small batches and trunk-based development keep merge costs low and let teams trace a regression to a single change. The trunk-based development reference describes the practice as a key enabler of continuous integration, with short-lived branches and every team member committing to trunk at least once every 24 hours, after a full local build that covers compilation, unit tests and integration tests.2 Feature flags keep deployment and

release apart, which allows teams to deploy unfinished work safely.

Human review time belongs to design and risk. Linters and formatters should settle style before a reviewer sees the diff.

Framework choice is a quieter lever. Rails' convention-over-configuration approach is a big reason companies like Shopify and GitHub could move quickly early on, and it is why teams scaling a Rails codebase often look at specialized Ruby on Rails development companies rather than generalist vendors when they need extra capacity without losing those conventions.

## Building a testing strategy that does not become a bottleneck

A healthy suite rests on fast unit and integration tests, with a small and stable end-to-end layer around the journeys that generate revenue. Hundreds of brittle user interface tests do the opposite. Automate regression on stable, high-traffic flows first, and leave exploratory testing to people. Shift-left means that quality assurance [QA] engineers check acceptance criteria before code is released, not after code freeze.

Teams should tolerate zero flaky tests. Once engineers learn to rerun failures until they pass, the suite stops catching anything. Google's testing team measured the scale of the problem: about 1.5% of all test runs reported a flaky result, almost 16% of tests showed some level of flakiness and about 84% of observed pass-to-fail transitions involved a flaky test.3 Its mitigations included rerunning failed tests, marking a test as flaky so that it reports failure only after three failures in a row and quarantining tests with excessive flakiness while filing a bug. The same post concedes that quarantine can mask real bugs, which is why zero tolerance remains the safer target.

When release cadence outgrows the in-house QA team, the usual answers are hiring, which is slow, or bringing in outside help. Teams going the second route tend to compare software testing companies on domain experience and automation maturity rather than headcount, since a partner that only adds manual testers just moves the bottleneck.

## How regulated and high-risk domains change the equation

A defect can become a patient safety event or a violation of the Health Insurance Portability

and Accountability Act [HIPAA]. The HIPAA Security Rule requires covered entities and business associates to put in place appropriate administrative, physical and technical safeguards to ensure the confidentiality, integrity and availability of electronic protected health information.4 Software under Food and Drug Administration [FDA] control also needs traceable requirements, tests and releases. Interoperability raises the bar further: a Fast Healthcare Interoperability Resources [FHIR] integration fails if it cannot find a unit of measure, or if a partner system does not read the specification exactly as written.

The HealthTech teams that keep shipping weekly under HIPAA and FDA scrutiny usually do it by automating the compliance burden itself, generating test evidence and traceability as a by-product of the pipeline. Some build that capability internally, while others rely on healthcare test automation services that already understand validation requirements and FHIR-based integrations. Either way, regulation forces a discipline that most product teams adopt only after a painful incident.

## Measuring whether you have the balance right

DORA's metrics work as a balanced scorecard. Change lead time and deployment frequency track speed, while change fail rate and recovery time track stability. The program now lists five metrics: change lead time, deployment frequency, failed deployment recovery time, change fail rate and deployment rework rate. It also states that speed and stability are not tradeoffs, and that top performers do well across all five.5

Leading indicators surface problems earlier. Useful ones include pull request cycle time, build time, flaky test rate and the share of each sprint spent on unplanned work. Over-indexing on speed shows up as a rising incident count and a hotfix culture. Over-indexing on quality shows up as too many reviews, too many approvals and gold-plated features.

Report quality work to non-technical stakeholders in terms of delivery. If unplanned work falls from 30% to 15% of sprint capacity, the planning deck should show that change in plain numbers. Treat the metrics as team-level signals for improvement rather than individual targets, because targets on individuals invite gaming of the numbers.

- 1Technical debt quadrant
- 2Trunk based development
- 3Flaky tests at Google and how we mitigate them

- 4The Security Rule
- 5DORA's software delivery performance metrics

## Summary

The right balance between speed and rigor shifts as a product ages. A team at seed stage testing demand can keep more two-way doors open than a company whose thousands of customers pay for its product. Review the balance every quarter against the same criteria, and treat each new piece of intentional debt as an item that someone has agreed to repay. Keep the pipeline fast, hold flaky tests at zero and match rigor to the cost of a defect. Regulated work narrows the flexible zone, and delivery metrics show whether the team's choices are holding. Teams that read these signals on a fixed schedule catch drift before it turns into an incident.