

Why the third change costs more than the first

The first change was easy and the second was fine. By the third, the estimate has doubled and nobody can say exactly why.

Author	Amit Govil, Founder and CEO, XONIK
--------	------------------------------------

Published	October 22, 2026
-----------	------------------

Reading time	8 min read
--------------	------------

Online	https://xonik.com/insights/perspectives/why-the-third-change-costs-more-than-the-first/
--------	---

IN BRIEF

The third change costs more than the first because each change leaves a reader, a rule or an exception behind, and the next change must first find out what those leftovers touch. That finding-out grows with every change that went unrecorded. Written decisions, tests, declared readers and scheduled removal flatten the climb.

KEY FINDINGS

1. Every change leaves residue that the system accepts without recording.

A reader, a rule, an exception or a workaround: none is large, each was agreed to, and nowhere does anything say why. A workaround that stays long enough becomes a feature that a downstream team builds its routine on.

2. An undeclared reader does the worst damage.

Any output that is easy to read will be read by someone who did not tell you, and that hidden coupling raises the cost of changing the producer, even to improve it.

3. The task stays the same size, and the work of finding out what it touches grows.

Each change must avoid the residue of the last, memory fades, and some residue belongs to people who have left. Unknowns are treated as risk, which is the margin a sponsor sees when an estimate jumps.

4. The usual protections add a fixed toll to every later change.

Manual checks, reviews, freezes and larger releases each answer a risk the team can feel. A check costs the same whatever the size of the change, and grouping makes each release harder to reason about.

5. Declared readers, written reasons, tests and scheduled removal flatten the climb.

Removal is the habit that is skipped, because it has no deadline and no sponsor, and it repays best. The argument is not against coupling but for declaring it, since declared coupling can be planned for.

What does a change leave behind?

Every change is made to solve a problem in front of a team, by the quickest route that is safe that day. The route leaves residue. A reader: another report, job or partner now depends on a field the change touched. A rule: orders from one channel are treated differently. An exception: except for this account, for reasons that were good at the time. A workaround: a manual step added until the proper fix, which stays because the fix never arrives. None of them is large, and each was agreed to. What they share is that the system accepts them without recording them. The code runs and the report reads, and nowhere does anything say why.

The reader does the worst damage. A paper by Google engineers on hidden technical debt describes undeclared consumers: systems that silently use another system's output as their input. The authors say this creates a hidden tight coupling that can sharply raise the cost and difficulty of making any change to the producer, including an improvement, and they tie it to what classical software engineering calls visibility debt. The paper concerns machine learning systems, but the mechanism is not specific to models. Any output that is easy to read will be read by someone who did not tell you.

Residue also hardens. A workaround that stays long enough becomes a feature: a team downstream builds its month-end routine around the manual step, and removing the step later breaks a process that nobody on the changing team knew existed. The longer a leftover lives, the more of the business is standing on it, and the more a clean-up looks like a change of its own.

Why does each change cost more than the last?

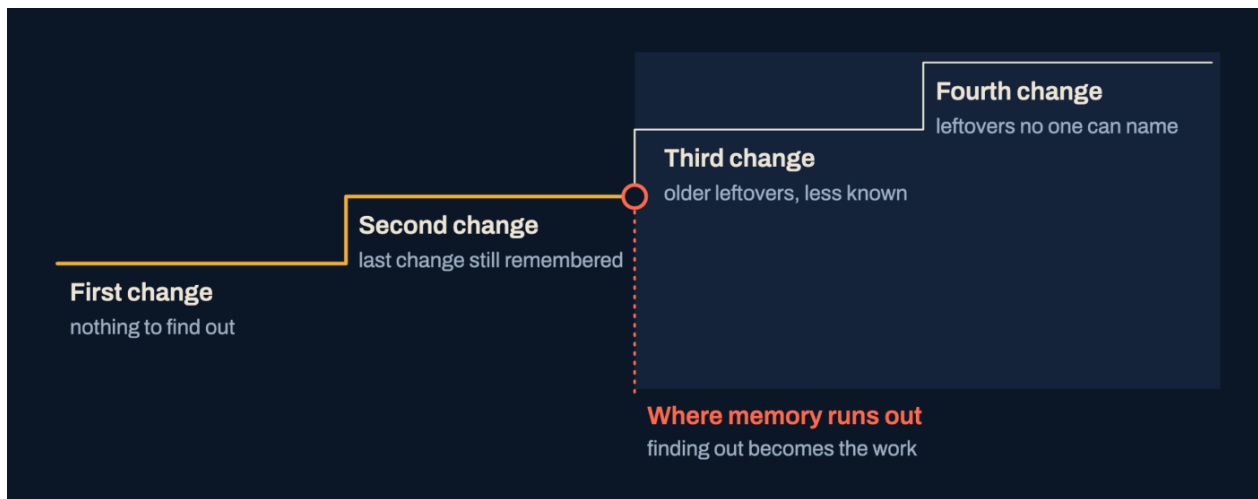
The work of the change itself stays about the same size. What grows is the work of finding out what it touches. The first change meets a system with nothing left behind. The second has to avoid the first one's residue, and the team still remembers it. The third must account for both sets, and one of them has begun to fade from memory. By the fourth, some of the residue belongs to people who have left. Exhibit 1 draws the stair. Effort rises not because the task is harder but because the question of what else uses this has less and less of an answer on paper.

Further effects make the stair steeper than a simple sum. Residue interacts: a rule for one channel and an exception for one account meet in the case of that account on that channel, and every new item can meet every old one, so interactions multiply faster than items. And every unknown has to be treated as a risk, so the estimate carries a margin for what nobody can see, and the margin is wider where less is written down. That margin is what a sponsor sees when the estimate jumps and nobody can say why.

D. L. Parnas made the same point in 1972 with a small example. In one way of dividing a system, a change to where the data was stored forced changes in every module. In another, each module hid one design decision from the others, and the same change stayed inside one module (On the Criteria To Be Used in Decomposing Systems into Modules, Communications of the ACM, December 1972). The cost of a change is set by the number of places that know about the decision being changed.

EXHIBIT 1

What the next engineer must find out



The height of each step is not the size of the task. It is how much must be discovered before the task can begin.

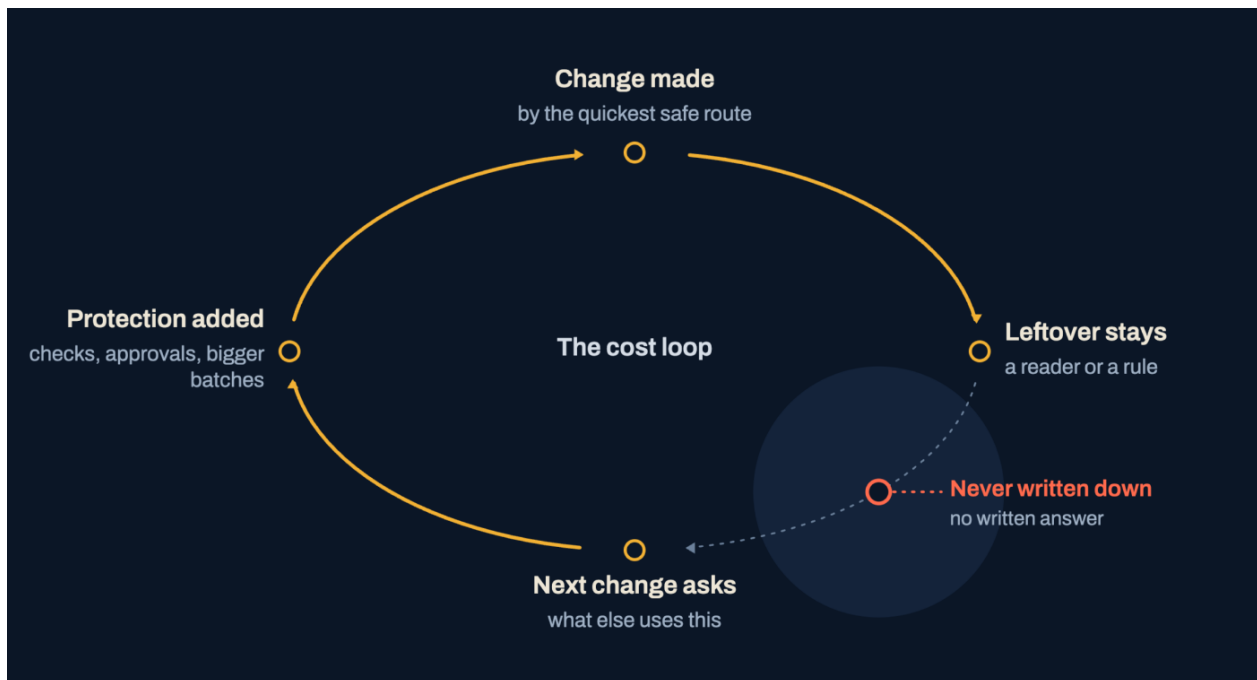
How do teams respond, and does it help?

Teams respond sensibly. They add a manual check before release, a review by the person who knows the old behavior, a freeze near busy periods, and a habit of grouping changes into larger releases so that the cost of checking is paid once. Each is a reasonable answer to a risk the team can feel. Each also adds to the cost of every later change, because a check is a fixed toll on every change whatever its size, and grouping makes each release bigger and harder to reason about.

Exhibit 2 shows the loop. A change leaves a leftover. The leftover is not written down. The next change has to ask what else uses it. The answer is protection, and the protection makes the next change slower and larger, which leaves more behind. The break is at the step where the residue goes unrecorded, because everything after it follows from a question with no written answer.

EXHIBIT 2

The loop that raises the cost



Protection treats the symptom of the missing answer, so the loop runs again with a higher fixed cost on every change.

What does this look like in a business?

A retailer's delivery rules begin simply: a charge applies below an order value. The first change adds a waiver for loyalty members, made in the checkout. The second adds a marketplace channel that must never show the waiver, handled by a condition on the channel. The third request is a new offer for members who buy through the marketplace. Nobody can say, without reading the checkout code and asking finance, what the waiver and the channel condition do together, which report reads the delivery charge, or whether the customer-service tool recalculates it. The developer spends the first days of the estimate answering those questions, and the answers live in two heads and a spreadsheet. By the fourth request the team has stopped giving firm dates, and the business reads that as a slow team.

Had each change been recorded with its leftovers, the third estimate would have started from a short list and ended on a date. The list would have said that the waiver is read by the finance report and by the checkout, that the channel condition applies before the waiver, and that the marketplace feed ignores both. The development effort was never the large part. The large part was reconstructing, in the third change, what the first two could have written down in an afternoon.

What made the third change expensive was reconstructing what the earlier changes could have written down in an afternoon.

What flattens the climb?

These habits are each aimed at one kind of residue. Declare readers: an output that others may use is published with a named owner and a list of who reads it, and access that is not declared is not given. The Google paper recommends access restrictions or strict service-level agreements for exactly this reason, since engineers will use the most convenient signal at hand when a deadline presses. Write each rule where the next person will look, with its reason, in the same change that introduces it. Turn every rule and exception into a test, so the system itself states what it must still do and a change that breaks one fails before release. And retire: give each exception an owner and a review date, and treat its removal as scheduled work.

Removal is the habit that is skipped, because it has no deadline and no sponsor. It repays best. The same paper lists deleting dead code and reducing dependencies among the ways debt is paid down, and recommends re-examining experimental branches in production code to see what can come out. Each removal shrinks the set of things that every later change must consider.

This is a description of a mechanism seen in delivery work and in the two papers cited. It measures nothing, and it does not say that a system should never be coupled. Some coupling is the purpose of a system. The argument is that coupling should be declared, because declared coupling can be planned for and undeclared coupling has to be discovered.

Sources read for this piece

Both papers were read in full on 20 October 2026. The climb and the loop come from delivery work and are offered as a mechanism. Nothing is measured.

1. [D. Sculley and colleagues, Hidden Technical Debt in Machine Learning Systems, Advances in Neural Information Processing Systems 28, 2015: undeclared consumers, visibility debt, ways of paying down debt](#)
2. [D. L. Parnas, On the Criteria To Be Used in Decomposing Systems into Modules, Communications of the ACM, volume 15, December 1972: effect of a change under two decompositions](#)

USE THIS

[Use the digital transformation roadmap template](#)

[Read why systems and data stop agreeing](#)

For the next change you make, write down what you had to find out first. That list is the documentation the previous change should have left.

This document was made from the article at

<https://xonik.com/insights/perspectives/why-the-third-change-costs-more-than-the-first/>. The online article is the current version.