// ARTICLE · PUBLISHED 1 NOVEMBER 2026 · UPDATED 2 NOVEMBER 2026

Pareto in Practice: Eighty Percent from Three

Two distributions measured inside one repository, one far sharper than the rule and one nowhere near it, and what each one licenses

Jason Newell · MAX Research Collective · Article ART-CLD-011 · Published 1 November 2026
Licensed under the Creative Commons Attribution 4.0 International License (CC BY 4.0), https://creativecommons.org/licenses/by/4.0/
Original: https://www.maxresearchcollective.com/articles/pareto-in-practice.html
Cite as: Newell, J. (2026). Pareto in Practice: Eighty Percent from Three. MAX Research Collective, ART-CLD-011. https://www.maxresearchcollective.com/articles/pareto-in-practice.html

The rule is a finding you measure

The 80/20 rule gets quoted as a planning heuristic. It's a property of a distribution, and you either measured it or you didn't.

The folklore drops three things. It's an observation rather than a law, so some distributions are skewed and some are flat, and "80/20" is a finding somebody made rather than a licence anybody holds. The two numbers need not sum to a hundred: ours come out at 80/3 for one population and 53/19 for another. And it shows where the mass sits rather than what matters, because a test that runs in four tenths of a millisecond can be the one that catches the outage.

"Focus on the 20% that matters" becomes a plan only when somebody says which 20%, and how they measured it.

Both populations come from one repository, and every number comes off this tree.

The sharp one: where the clock goes

The backend suite holds 14,078 tests and runs 24.0 minutes serially; the median test finishes in 11.2 milliseconds. The slowest one takes 153.5 seconds — roughly 13,700

times the median.

Sort by duration. Accumulate. The shape arrives before you have finished reading the first few rows, and it isn't the shape the rule predicts.

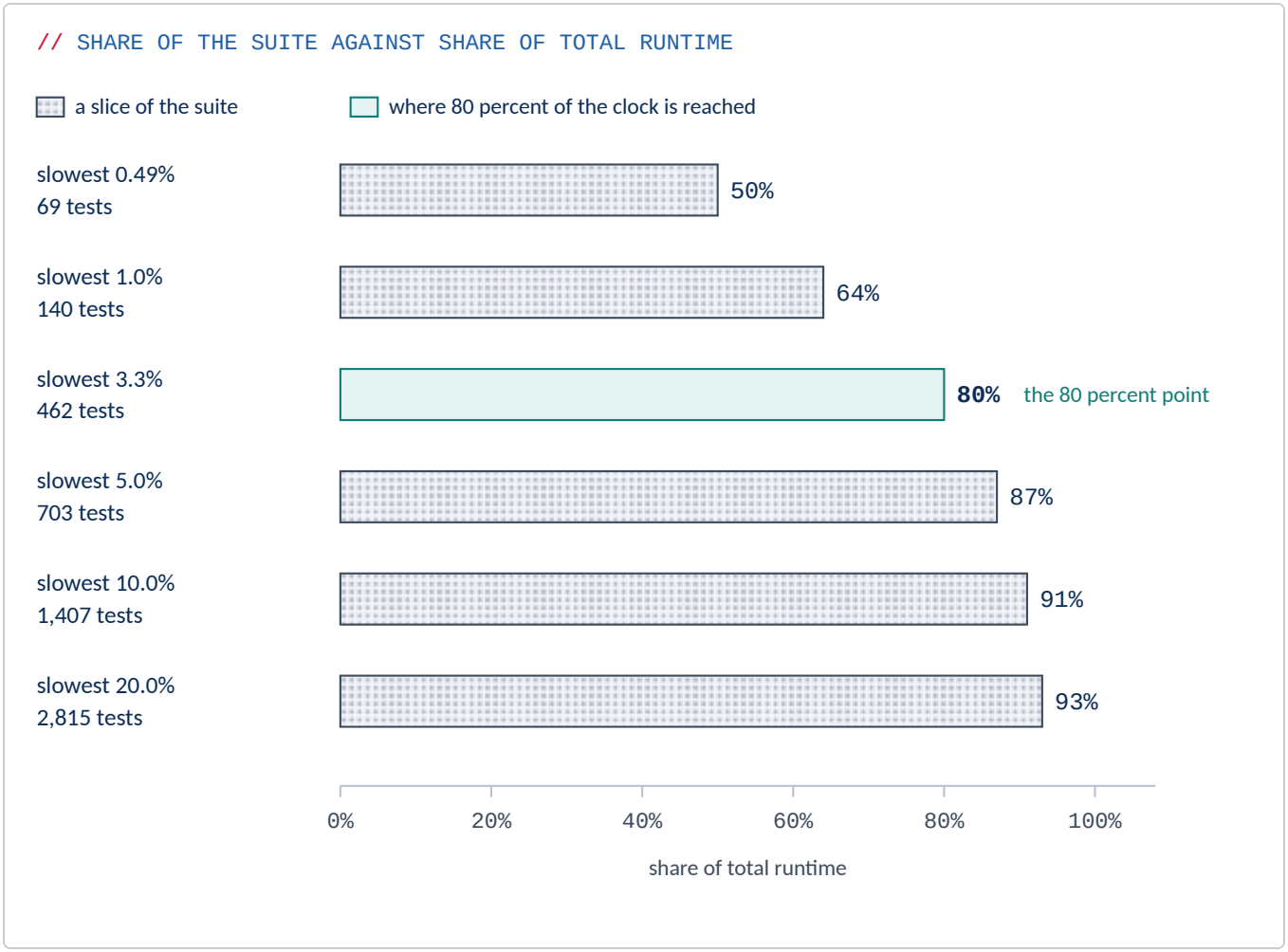


Figure 1. Share of the suite against share of total runtime, sorted slowest first. The marked point is where 80 percent of the clock has been accounted for.

Eighty percent of the runtime comes from 3.3 percent of the tests. Call it 80/3. The difference between that and 80/20 is the difference between a project and an afternoon: 462 tests to examine instead of 2,815. A suite-wide push to make the tests faster would spend most of its effort on the 96.7 percent that jointly cost a fifth of the clock.

It also means the median lies, and lies badly. Half the suite finishes under 11.2 milliseconds; one test runs for two and a half minutes. Any decision made on the

average test is wrong about nearly every actual test.

One level higher, where the tests are grouped by what they prove, the same sharpness shows up again.

TIER	SHARE OF RUNTIME	WHAT IT PROVES
Unit	75.0%	Logic, in isolation
System	11.4%	Contracts, wiring, and the pins that may not move
Integration	8.6%	Seams between real components
Route contracts	5.0%	What the interface promises
Topology	Under 0.1%	That the ecosystem graph is consistent

Table 1. Runtime by tier. One tier is three quarters of the bill.

One tier carries three quarters of the cost. That single row is what makes the draft-scoping decision worth having — and it's also what made the version we got wrong look so attractive.

The flat one: where the defects come from

The defect registry is an append-only record of bug classes that already shipped here and already failed. Nothing ever leaves it. If Pareto held, a few families would dominate, and the sensible move would be to pick those few and harden them until the recurrence rate fell.

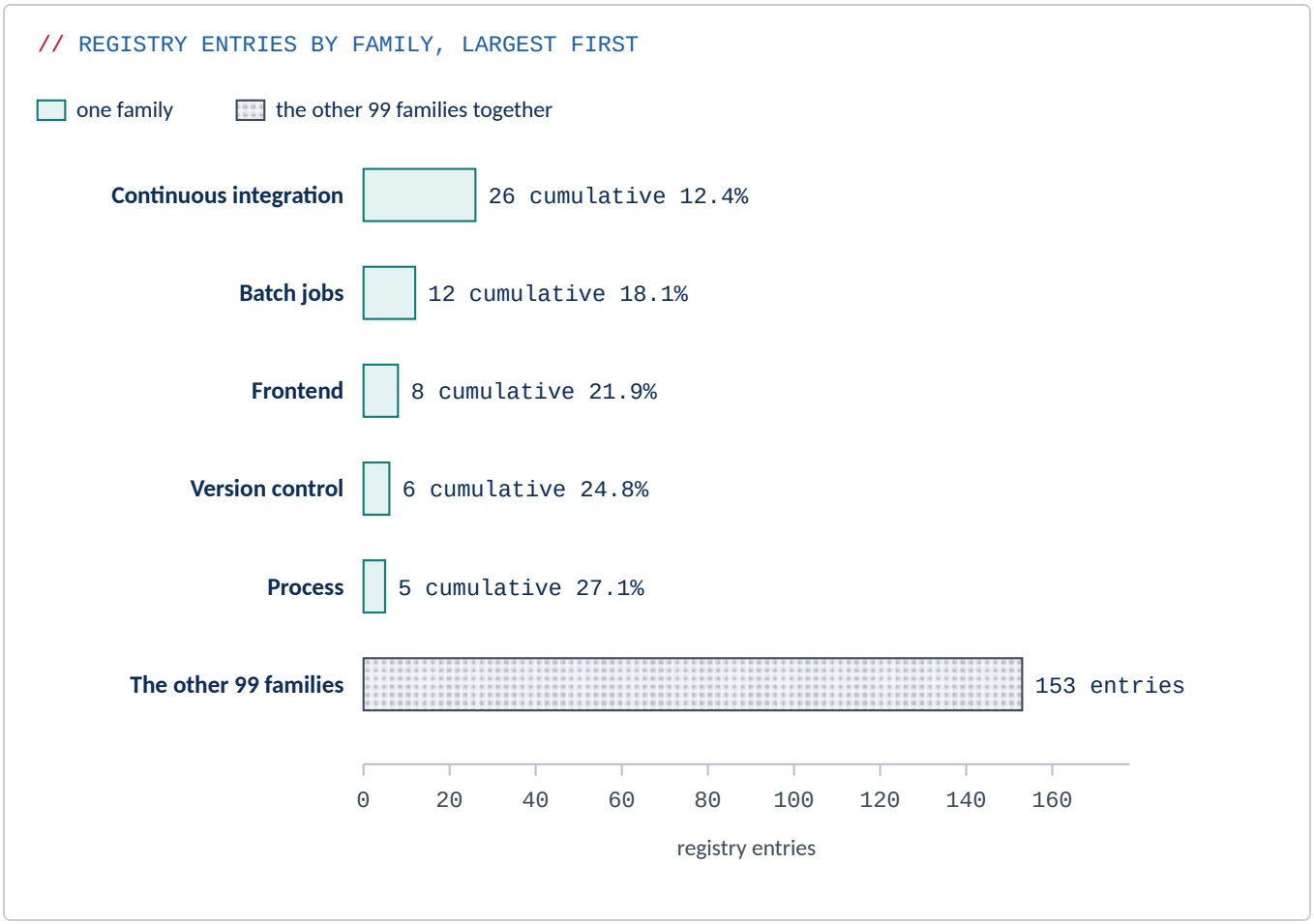


Figure 2. Registry entries by family, largest first. The tail is 99 families, and most of them hold one entry or two.

Two hundred and ten entries spread across 104 families. The largest accounts for 12.4 percent of the registry, and 96 of the 104 hold one entry or two. Nothing dominates. There's no single cause available to eliminate, which is an unwelcome answer and still the answer.

POPULATION	WHAT THE TOP 20% ACCOUNTS FOR
Test runtime, 14,078 tests	93%
Defect classes, 210 entries	53%

Table 2. The same rule tested against two populations in one repository.

Measuring both populations is what made the real point visible. Pareto is an understatement for runtime and simply false for defects. Had we assumed 80/20 for bugs (the way a roadmap usually does) we would have built a few deep defences. The evidence calls for the opposite: broad, cheap, mechanical gates — a lint that sweeps 1,509 files, a documentation check, a manifest check, one syntax-level pin per contract.

So the rule that survives both cases is a conditional rather than a ratio, and it fits on one line. Where the distribution is sharp, concentrate. Where it's flat, automate cheaply and broadly. Measure first, because the two answers are opposites and picking the wrong one costs a quarter of a year.

Spending the leverage on the build

Sharding buys latency rather than cost

The unit suite splits four ways by recorded duration rather than by file count.

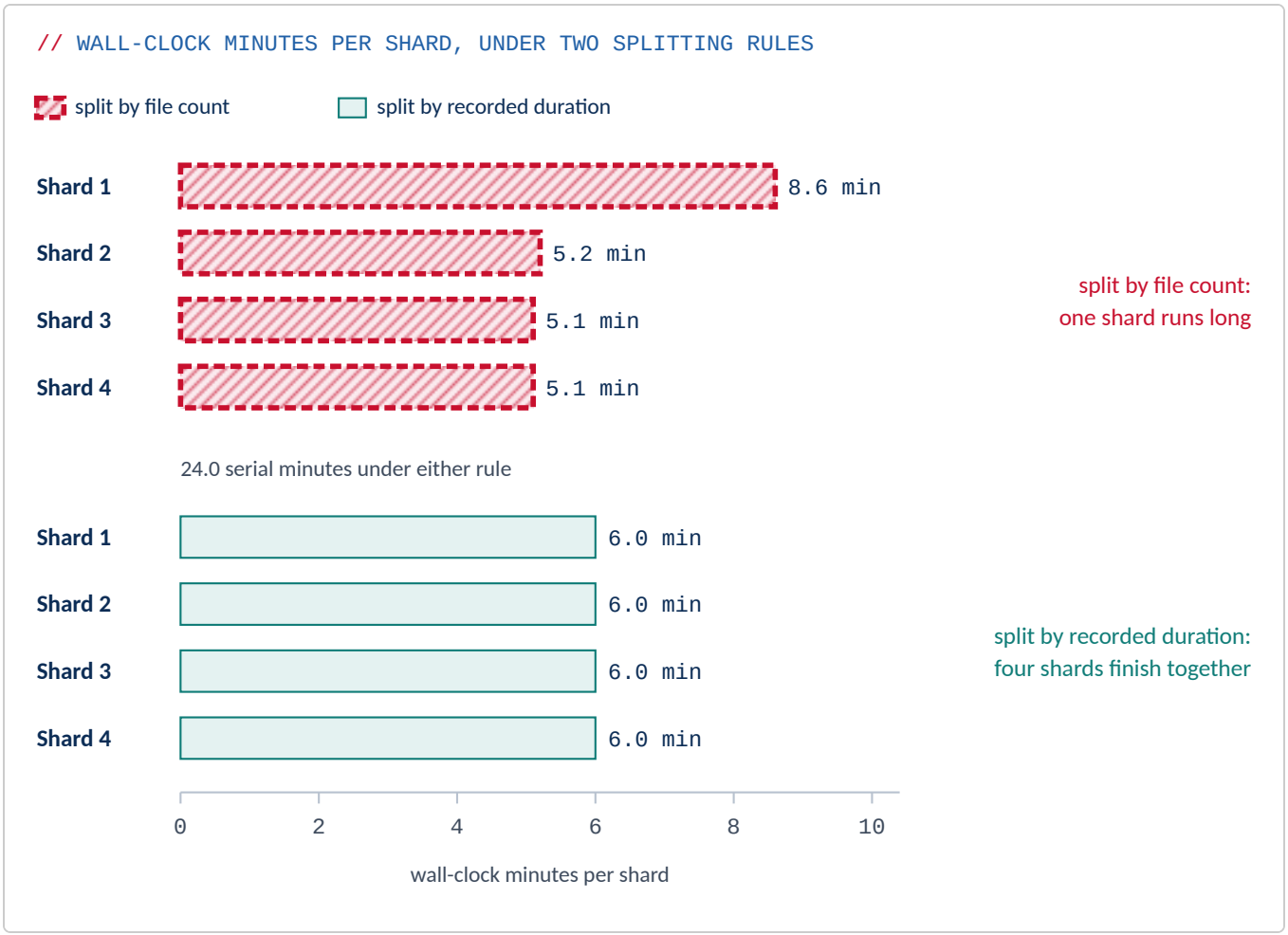


Figure 3. Twenty-four minutes of serial tests split four ways, by file count and by recorded duration.

Splitting by count puts roughly 3,500 tests in each shard and drops the 153-second test beside 3,499 others; one shard then runs long while three sit idle. Splitting by duration packs the heavy 3.3 percent deliberately, so the four shards finish together. That's the Pareto-aware choice, and it exists only because somebody recorded the durations first.

Sharding doesn't make the suite cheaper; four shards cost the same machine minutes plus four times the fixed setup. What they buy is latency (the thing a working session actually waits on).

Tier scoping, and the part we got wrong

A draft change runs a reduced set of tiers, and the full pyramid fires when the change is marked ready.

DRAFT SCOPE	TIERS RUN	DRAFT TEST TIME SAVED	STATUS
Full	Unit, integration, routes, topology, system	None	Available on request
Unit	Unit and system	About a quarter	The current default
Contract	System only	About seven eighths	Retired

Table 3. Three draft scopes, and the one that got retired.

Contract was the aggressive play: drop seven eighths of the test time and keep the floor. On the arithmetic it's the obvious winner — and the arithmetic was measuring the wrong thing.

Under that scope neither the unit tier nor the integration tier ran on any draft push. One change spent a whole session in that state. It carried four broken fixtures, three failing unit tests and a two-shard hang through three green draft checks, and every one of them surfaced only on the ready-flip pyramid, cycles later than a unit-tier draft run would have caught them.

The lesson generalises well past a build configuration. A Pareto cut optimises the metric you measured, and we had measured minutes. What mattered was minutes until a break becomes visible. By that measure the cheap gate was the expensive one, because a defect found three cycles late costs a rebuild, a re-review, and a cold-start diagnosis by somebody who has since loaded another problem into their head.

The unit tier is back on drafts: 75 percent of the test time and, on this evidence, most of the signal. Integration, routes and topology still wait for the ready flip, and that part of the cut survived, because nothing in the record shows it hiding defects the same way.

The floor the rule may not touch

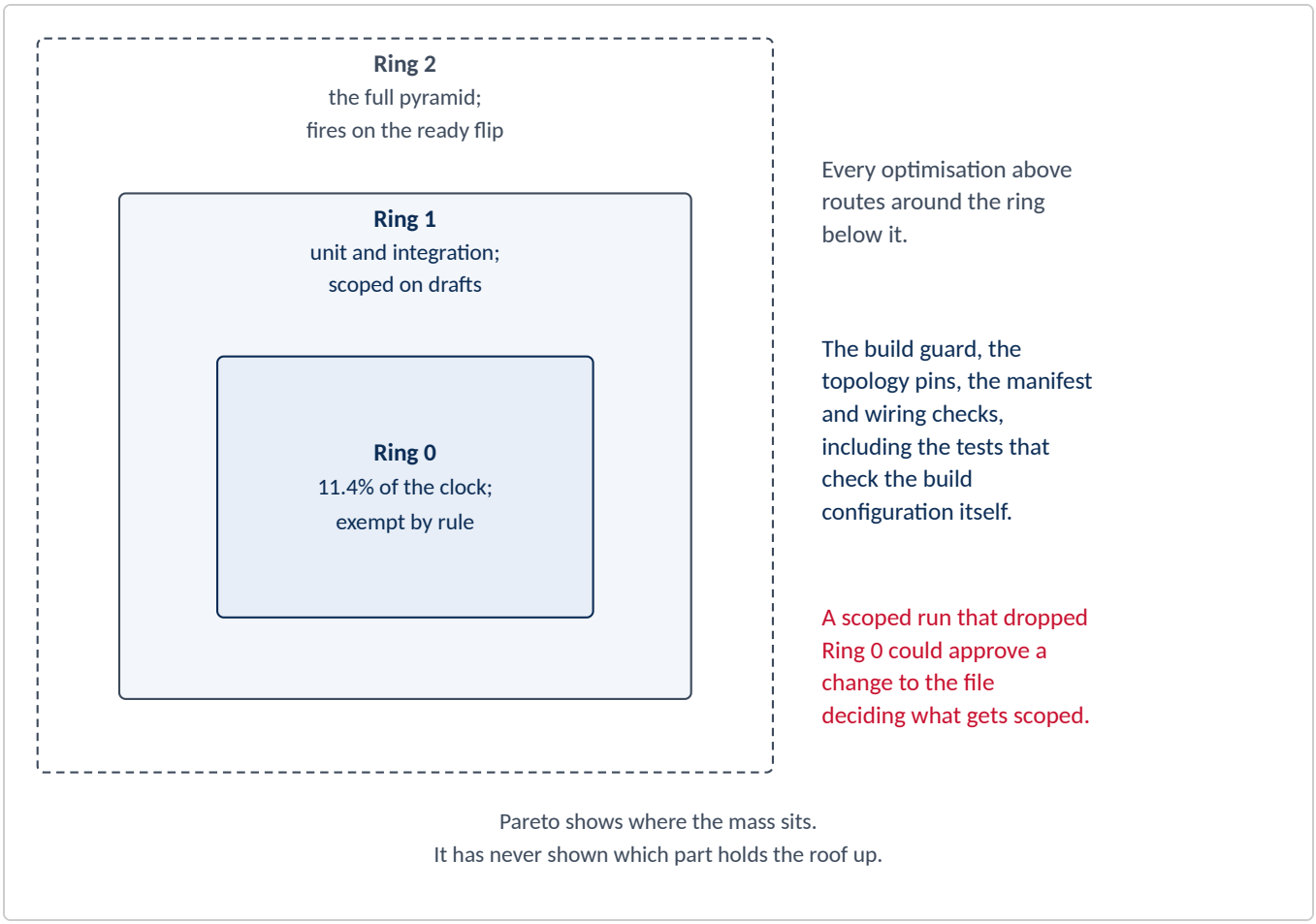


Figure 4. The three rings, and the one that every optimisation has to route around.

Ring 0 is 11.4 percent of the clock and exempt from every optimisation, by written rule. It holds the build guard, the topology pins, the manifest check and the wiring checks, including the tests that check the build configuration itself.

Read that last clause twice; it's the whole reason for the exemption. A scoped run that dropped Ring 0 could green-light a change to the very file deciding what gets

scoped. Some 20 percent is load-bearing. Pareto shows where the mass is rather than which parts hold the roof up.

Coverage, and how it invites abuse

Coverage turns into self-deception faster than any other number here, because it moves whether or not anything was verified. An inert assertion executes perfectly and reports full coverage. Two habits keep it honest.

The first habit is mutation verification. A test earns its place only once breaking the code breaks the test. From the change that prompted this article: neutering one six-line guard failed five of six new tests, and the sixth was the control proving the healthy case still passed. Without that check the six prove only that they run.

The second is fringe detection. Rather than chase a percentage, the build computes the blast radius of a change and asks one question: is any changed symbol reached by no test at all? That aims straight at the zero-coverage tail, which is where the risk actually concentrates.

ANTI-PATTERN	WHY IT FAILS	INSTEAD
Cover the 20% of code with 80% of the bugs	It needs to know where the bugs are, and the measurement above says they are diffuse	Broad cheap gates, with the claim mutation-verified
Chase a global coverage percentage	A high average hides a zero-coverage fringe	Ask which changed symbols have no test
Delete slow tests to hit a budget	Slow and valuable are uncorrelated	Shard by duration, and keep the test

Table 4. Three ways a coverage number gets used against the people relying on it.

The same shape at runtime

The logic runs on in the live system. An always-on service keeps a roster of health probes under a Pareto-shaped constraint. A check that fires constantly gets disabled inside a week, so the roster has to report healthy for the designed quiet state and spend its alerting budget on genuine divergence.

A microphone probe is the worked example. A closed microphone is the correct state most of the time — the system arms it only when a client connects — so "microphone closed" can't be the alarm.

SIGNAL	MEANING	ALARM ON ITS OWN?
Listening wanted	The system is supposed to be listening	No
Microphone open	The manager believes the stream is open	No
Capture running	The stream's own view of itself	No
Wanted and open disagree	Something failed in silence	Yes

Table 5. Four signals, of which only the disagreement is worth waking somebody for.

That divergence is the 20 percent worth a person's attention, and it stayed invisible for months for exactly the reason the table shows. Each signal, read alone, looked entirely normal.

Why it matters

WHERE	THE DISTRIBUTION	VERDICT	WHAT IT LICENSES
Test runtime	80% from 3.3%	Sharper than the rule	Duration-based four-way sharding
Tier cost	Unit is 75%	Sharp	Scope drafts, then partly reverse it
Defect classes	Top 20% is 53%	The rule does not hold	Broad gates rather than deep ones
Ring-0 checks	11.4% of runtime	Irrelevant to the question	Exempt by written rule
Coverage	A long zero-coverage tail	Sharp, and easy to fake	Mutation verification and fringe detection
Runtime probes	Few true divergences	Sharp	Alert on divergence rather than on state

Table 6. Six places the rule was tested, and what each measurement licensed.

Skipping the measurement has a price, and it lands on the team that skipped it.

A team that assumes 80/20 for its defects builds a few deep defences and leaves a hundred small families open. A team that cuts its build on minutes per run ships breaks that surface three cycles late, after the author has moved on to something else. And a team that never writes down its exempt fraction will one day watch a reasonable-looking optimisation remove the checks that would have caught it.

The rule tells you where the weight is. It has never told anybody which part is holding the roof up.

Newell, J. (2026). MAX Research Collective, ART-CLD-011. CC BY 4.0

Glossary

TERM	DEFINITION
Blast radius	The set of symbols a change can affect, computed from the code rather than guessed. Fringe detection asks which part of it no test reaches.
Coverage	The share of code a test run executed. It says nothing about whether the execution would have failed had the code been wrong.
Draft and ready	Two states of a proposed change. A draft runs a reduced set of test tiers; marking it ready fires the full pyramid.
Fringe detection	A check that names the changed symbols no test reaches, in place of chasing a coverage percentage.
Mutation verification	Deliberately breaking the code a test claims to guard, and requiring the test to fail. A test that stays green has left the job without leaving the roster.
Pareto distribution	A distribution where a small share of members accounts for most of the total. Whether one applies to a given population is a measurement rather than an assumption.
Ring 0	The set of checks exempt from every optimisation by written rule, including the checks that verify the build configuration itself.
Sharding	Splitting one test suite across several machines running at once. It reduces wall-clock latency and leaves total machine time roughly where it was.
Tier	A group of tests defined by what they prove: unit for logic in isolation, integration for the seams between components, system for contracts and wiring.

CLOUD / DEVOPS ART-CLD-011

// ABOUT THE AUTHOR

Jason Newell leads organizations and people in technology, and is a researcher, deep technologist and builder — from a CRAY to agentic AI. He founded MAX Research Collective to build AI for regulated industries that runs where its owner has control, with healthcare as its strongest focus. About Jason → Connect on LinkedIn →

Newell, J. (2026). MAX Research Collective, ART-CLD-011. CC BY 4.0