Qase

The Economics of Testing

How much testing do we need?

From risk to investment in 4 steps

Executive Summary

Most organizations treat testing as a cost to endure rather than an investment to optimize. The result is predictable: either too much testing in the wrong places, or too little where it matters most. Both waste money.

This whitepaper presents a four-step process for making testing decisions based on economics rather than gut feeling. It starts from the risks that actually threaten your product, quantifies their potential impact, and builds a testing portfolio that targets the highest-exposure risks first. It then shows how to review and rebalance that portfolio over time so it stays aligned with reality.

The core reframe: your organization is already paying for quality. Most of that spending goes to the most expensive bucket: failure costs (incidents, support, hotfixes, lost revenue). The question is not whether to invest in testing, but where to redirect spending so you pay less overall while catching more problems earlier.

The approach is grounded in ISO/IEC 25010 (quality characteristics), ISO/IEC/IEEE 29119 (risk-based testing), and the Cost of Quality (CoQ) framework. It is designed for practitioners: QA leads, test managers, and engineering managers who need to justify their testing strategy with more than ‘we need more coverage’.

A single running example (payment processing failures due to regression defects) threads through all four steps, showing how the same risk moves from a vague fear to an explicit statement, gets prioritized by exposure, drives specific testing and prevention decisions, and is reviewed for cost-effectiveness over time.

“How much testing?” is the wrong question

Every testing team hears this question eventually. It sounds reasonable. It is not.

“How much testing do we need?” assumes testing is a quantity problem: more is better, less is worse. That framing leads to arbitrary answers like “80% code coverage” or “two weeks of QA before release”. These numbers mean nothing without knowing what is at risk.

Consider two products:

- **Medical device software** that controls insulin pump dosing. A defect can kill a patient. Regulated by FDA. Must comply with IEC 62304. Requires formal verification, MC/DC coverage, independent validation.
- **A corporate CMS** for an internal blog used by 50 employees. A defect means a blog post has wrong formatting. Nobody dies. Nobody gets fined.

Asking “how much testing?” for both products is meaningless. The medical device needs exhaustive safety testing and regulatory evidence. The CMS needs a smoke test and maybe some manual checks. The difference is not that one team is more serious about quality. The difference is that the risks are fundamentally different.

Testing is not a quantity to dial up or down. It is an investment portfolio. You select a mix of activities (different types, levels, techniques) that provides the maximum risk reduction for the resources available. Like any investment, it has diminishing returns: the first tests catch the most obvious problems, and each additional test catches progressively rarer issues. At some point, the cost of the next test exceeds the expected value of the risk it covers.

This is comparable to insurance or preventive healthcare. We do not buy scuba diving insurance for our daily commute because we do not face those risks. But construction workers invest in expensive safety equipment and additional insurance because the risks are real and the consequences are severe. The investment decision follows from the risk, not from a generic 'amount'.

The right question is not **"how much?"** but **"which risks justify which investments?"**

KEY TAKEAWAY

Stop asking "how much testing?" and start asking "which risks justify which investments?" The answer depends on your product, your context, and what failure would actually cost.

Your organization is already paying for quality

Before diving into the four steps, there is a reframe that changes how you talk to management about testing.

The Cost of Quality (CoQ) framework says all quality-related costs fall into four buckets:

1. **Prevention costs:** spending to stop defects from being introduced (training, design reviews, coding standards, threat modeling, architecture reviews, pair and mob programming)
2. **Appraisal costs:** spending to detect defects before customers are impacted (testing, code reviews, static analysis, quality audits)
3. **Internal failure costs:** spending caused by defects found before release (rework, debugging, re-testing, delayed releases)
4. **External failure costs:** spending and losses caused by defects found after release (incidents, support, lost revenue, fines, reputation damage)

These are not abstract categories. They are real costs that appear in real budgets. And in most organizations, the picture looks the same: the bulk of quality spending goes to failure costs. Firefighting incidents. Customer support. Hotfixes. Rework. Very little goes to prevention.

That means you are paying the highest price for defects you could have prevented cheaply.

Here is the reframe that works with managers: this is not “we need to spend more on prevention”. This is “let us cut the failure costs”. Managers respond to cost-cutting.

They respond less well to arguments about prevention that sound like “trust me, this will pay off someday”. CoQ gives you the language to show what incidents actually cost and where the money is going.

Prevention has a visibility problem, though. When prevention works, nothing happens. No incident, no outage, no angry customers. Success is invisible. Nobody thanks the design review for the incident that never occurred. Medicine has the same problem: the biggest bills go to treating disease, while the vaccines and checkups that prevent disease are invisible. You just do not get sick as much. This is the counterfactual problem: you cannot directly observe what would have happened without your controls.

That visibility gap is exactly why CoQ matters. It makes visible what is otherwise invisible. You can point at real numbers: here is what we spent on incidents last quarter, here is what we spent on prevention, and here is the ratio. The numbers do not require faith.

Prevention is the cheapest lever per unit of risk reduction. A defect caught in a design review costs a conversation. The same defect caught in production costs an incident, customer support, hotfixes, maybe lost revenue, maybe regulatory consequences. External failure costs are almost always higher than internal failure costs, which are higher than appraisal costs, which are higher than prevention costs.

The four-step process that follows is how you decide where to redirect that spending.

KEY TAKEAWAY

Your organization is already paying for quality, mostly in the most expensive bucket (failure costs). CoQ makes that spending visible and turns quality improvement into a cost-cutting conversation.

Step 1: Find your most expensive risks

Start with fears, not test cases

Most teams start testing by writing test cases. That is backwards. Before you decide what to test, you need to know what could go wrong and what it would cost.

The input is not a requirements document. It is the fears and concerns that people across the organization carry in their heads. Managers worry about revenue and deadlines. Engineers worry about complexity and fragile integrations. Support worries about the same customer complaints every sprint.

Gather these fears from all relevant stakeholders:

- **Managers:** business impact, delivery timelines, customer satisfaction, compliance
- **Engineers:** technical debt, system complexity, integration risks, performance, security
- **Product/operations/support:** deployment risks, common failure patterns, user confusion

Use structured workshops, one-on-one interviews, retrospectives, and postmortems. The two mistakes to avoid: only asking engineers (you miss the business perspective) and rushing this step to get to ‘the real work’.

Convert fears into explicit risk statements

Fears are vague. ‘I am worried about performance’ is not actionable. Convert each fear into a structured risk statement:

Template: “If [condition/event], then [negative consequence] may occur, resulting in [impact on business/quality]”.

Running example: A team hears the fear “what if we break something in production?” and converts it:

> **"If a regression defect is introduced in the payment module, then transactions may fail silently, resulting in lost revenue and customer trust."**

The pattern: vague emotion becomes specific trigger, specific consequence, measurable impact.

Before moving on, each risk statement should pass three checks:

1. **Observable trigger:** can you detect when the condition occurs?
2. **Measurable consequence:** can you measure the negative outcome?
3. **Owner identified:** who is responsible for monitoring and responding?

Quantify potential impact

Once risks are explicit, size them using three-point estimation (optimistic, most likely, pessimistic). This makes uncertainty explicit rather than hiding behind false precision.

Ask engineers for effort/schedule impact: 'If this integration breaks, how long to investigate and fix?'

Ask managers for financial/business impact: "What is the potential revenue loss?"

Running example continued: For "payment processing failures due to regression defects":

- Engineering impact: optimistic 1 day, most likely 3 days, pessimistic 5 days to diagnose, fix, deploy, and verify (blocking the team)
- Financial impact: optimistic $20K per day, most likely $50K per day, pessimistic $100K per day in lost revenue until resolved

Why three-point and not a single number? Because single-point estimates hide uncertainty and create false confidence. When you later compare your estimates with actual outcomes in Step 4, you learn whether your ranges were realistic and can recalibrate. If your estimates are consistently off by 3x, that tells you something about your estimation process.

Two common mistakes: stopping at 'we do not know' instead of producing a rough range, and quantifying only one side (technical impact without business, or business without engineering speed).

Output of Step 1

By the end of this step you should have: a list of explicit risk statements (not vague fears), impact estimates as three-point ranges covering both engineering effort and business impact, and documented stakeholder agreement on which risks matter most. This is the concrete input for prioritization in Step 2 and investment decisions in Step 3.

KEY TAKEAWAY

Start from fears across the organization, convert them to explicit risk statements with measurable impact, and size them with three-point estimates. This becomes the foundation for every testing decision that follows.

Step 2: Prioritize by exposure

Likelihood times impact

With explicit, quantified risks from Step 1, you now rank them. The formula is straightforward:

Risk exposure = likelihood x impact

Score both on a 0-10 scale. Likelihood is the probability the risk materializes. Impact is the potential loss, using whatever criteria matter in your context: financial in commercial SaaS, safety in medical devices, legal in regulated industries.

Scoring is a decision aid, not a measurement of reality. The goal is consistent ranking and traceable rationale, not debates about whether something scores 7.2 or 7.4.

The punchline: boring risks often outrank scary ones

Running example continued:

Risk	Impact	Likelihood	Exposure
Code complexity slowing delivery	8 ($200K contract penalties)	9 (every sprint)	72
Payment module regression	7 ($50K/day)	8 (frequent changes)	56
Performance degradation	9 ($10K/hour)	5 (under load)	45
Security vulnerability (SQL injection)	10 ($2M+ fines and churn)	2 (rare, some controls)	20

Code complexity scores 72. The SQL injection vulnerability scores 20. A 3.5x difference. The boring, chronic risk that nobody's pager goes off for outranks the scary, catastrophic one.

This does not mean ignore security. It means code complexity gets investment first. Security might be acceptable with existing controls. Without the math, the loudest voice in the room decides.

Risk acceptance thresholds

Not every risk gets testing. Define thresholds:

- **Unacceptable** (exposure > 50): must be reduced through controls and testing
- **Acceptable with controls** (20-50): proceed if adequate controls are in place
- **Acceptable** (< 20): no additional investment needed

For any risk you accept above your threshold, document the rationale, assign an owner, and set a review date. 'We chose not to invest here' is a legitimate decision. 'We did not get to it' is not.

One more consideration: priorities shift as the product matures. In a prototype, functional suitability and usability dominate while maintainability carries less weight. In production, reliability, security, and performance rise. Revisit this ranking when the context changes.

KEY TAKEAWAY

Calculate risk exposure (likelihood x impact) for every risk. The ranking will surprise you: chronic, undramatic risks often outrank the dramatic ones. Use thresholds to make acceptance decisions explicit.

Step 3: From risks to testing decisions

With a ranked list of risks and their exposure scores, we finally talk about testing. But the testing landscape is enormous: types, levels, techniques, practices, static, dynamic, manual, automated. You need a systematic way to navigate it.

Quality characteristics: the bridge

The organizing principle is to map each risk to the quality characteristic it threatens. ISO/IEC 25010 defines eight product quality characteristics: functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability.

This mapping is the bridge from ‘what could go wrong’ to ‘what kind of testing addresses it’.

Running example continued: Our payment processing risk maps to **functional suitability** (transactions should work correctly) and **reliability** (the system should operate consistently). Once risks are categorized by quality characteristic, the testing menu shrinks. Reliability risks point to reliability testing. Security risks point to security testing. The mapping makes the connection explicit.

Risk	Quality Characteristic(s)
Payment module regression	Functional suitability, Reliability
Performance degradation	Performance efficiency
Security vulnerability	Security
Code complexity	Maintainability

The evidence ladder: cheapest evidence first

For each risk, consider controls in order of cost-effectiveness:

1. **Prevent via design and standards** (cheapest)**:** coding standards, architecture patterns, design reviews
2. **Detect via static review and analysis:** code reviews, SAST, linters
3. **Detect via unit and component tests:** fast, isolated, cheap
4. **Detect via contract and integration tests:** interface checks, API contracts
5. **Detect via system and end-to-end tests:** slower, more expensive
6. **Detect via production guardrails and telemetry** (highest consequence if missed)**:** SLOs, alerting, canaries, rollback

Select the lowest rung that produces credible evidence for the risk. Do not jump to expensive system tests when a unit test suffices. But do not rely on unit tests for risks that only manifest at system scope.

Many of the highest-return controls are not tests at all. Progressive delivery (canary releases, rollback automation), feature flags, runtime monitoring, and release gates are operational controls. Testing is the appraisal stream that produces evidence these controls exist and work.

Running example continued: For “payment processing failures due to regression defects”:

- **Prevention:** mandatory design reviews for all payment module changes, pair programming for critical paths
- **Static:** code reviews focused on payment logic, SAST rules for common payment error patterns

- **Unit/component:** automated unit tests for payment calculation logic (cheapest, fastest feedback)
- **Contract/integration:** contract tests for payment gateway API
- **System:** a small end-to-end happy path test for the complete checkout flow
- **Production guardrails:** transaction success rate monitoring, alerting on anomalies, rollback automation

This is a portfolio: multiple levels of evidence, from cheap to expensive, all targeting the same risk. The unit tests catch logic errors early and cheaply. The contract tests catch integration mismatches. The system test catches end-to-end flow issues. The production guardrails catch anything that slips through.

Selecting testing types, levels, and the static/dynamic balance

Testing types define what quality is being evaluated. Functional suitability maps to functional testing. Performance efficiency maps to performance, load, and stress testing. Security maps to security testing, penetration testing, and vulnerability scanning.

Test levels define where in the system hierarchy you produce evidence: unit, component, integration/contract, system, acceptance. The economic principle is to prefer the lowest-cost level where the risk can be detected with credible evidence. Different levels expose different fault classes:

- **Unit/component:** local logic errors, boundary conditions (fast, cheap)
- **Integration/contract:** interface mismatches, protocol violations
- **System:** end-to-end behavior, configuration issues, cross-cutting concerns
- **Acceptance:** suitability for users, operations, regulations

Static vs dynamic testing are fundamentally different ways of producing evidence. Static testing (code reviews, SAST, design reviews) can start weeks before dynamic testing and catches defects that are cheaper to fix. Dynamic testing (execution) is the only way to observe integration faults, concurrency issues, performance under load, and real UX problems. For critical risks, use both: static catches patterns early, dynamic validates real behavior.

Test design and coverage

Test design techniques define how you derive test cases. Higher-priority risks justify higher coverage:

- **High-priority risks:** comprehensive coverage (all equivalence classes, boundaries, decision rules)
- **Medium-priority risks:** moderate coverage (key scenarios, main transitions)
- **Low-priority risks:** basic coverage (happy path plus representative negatives)

Coverage is a proxy for thoroughness, not a guarantee of defect absence. Treat coverage targets as minimum evidence thresholds and validate them against actual outcomes in Step 4.

Choosing practices: how to execute

Test practices define how testing work is organized and executed. They are orthogonal to types, levels, and techniques. The key decisions:

Exploratory vs scripted. Exploratory testing is a practice, suited for discovery, ambiguity, and complex scenarios. Scripted testing suits regression, compliance, and repeatable CI/CD gates. Most teams need both: exploratory sessions for novel and uncertain areas, scripted automation for stable regression paths.

Manual vs automated. Manual testing works for exploration, human judgment, and one-off scenarios. Automated testing pays off for regression, CI/CD, and repetitive checks where reuse and frequency justify the setup and maintenance cost. The mistake is automating everything (judgment-heavy scenarios become brittle and expensive to maintain) or automating nothing (slow feedback, high labor cost for repetitive checks).

Delivery cadence. Match cadence to the risk profile: on-commit/CI for fast, cheap checks; nightly for broader or slower suites; pre-release for final validation; production/canary for gradual rollout with real telemetry.

Running example continued: For the payment processing risk, the team selects:

- Scripted automated unit and contract tests, running on every commit (CI gating)
- A nightly regression suite covering broader payment scenarios
- Monthly exploratory sessions focused on edge cases and unusual payment flows
- Production monitoring with transaction success rate alerts and rollback automation

The portfolio as a traceability chain

A testing portfolio is a traceable chain: risk leads to controls, controls produce evidence, evidence supports acceptance decisions, and review metrics validate whether the portfolio is working. Each testing activity should trace back to the specific risks it addresses.

This traceability is what turns ‘we do a lot of testing’ into ‘here is why we do each test, which risk it covers, and whether it is working’. It enables you to justify investments to management with specifics, not vague appeals to quality.

Overlap and coverage gaps

Overlap between testing activities is a feature, not a flaw: it provides safety margins. But only when the checks are diverse and low-correlation. Good redundancy: static analysis plus dynamic testing for the same risk, unit tests plus contract tests (different fault classes). Bad redundancy: the same functional check duplicated at unit, integration, and system level. Highly correlated checks add little safety and waste resources.

Coverage gaps are the real danger. Every high-priority risk must have at least one credible evidence stream. Review your prioritized risk list and flag any risk with no corresponding testing approach.

KEY TAKEAWAY

Map risks to quality characteristics, then select the cheapest credible mix of controls (prevention, static, dynamic, operational) that produces evidence for each risk. Ensure every high-priority risk has at least one evidence stream, and make overlap diverse rather than duplicated.

Step 4: Review and rebalance

A testing strategy is not a one-time plan. Like any investment portfolio, it must be reviewed to confirm it is still reducing the most important risks cost-effectively.

When to review

Build reviews into the normal lifecycle rather than waiting for a postmortem after something goes wrong.

- **Scheduled cadence:** per release, quarterly, or whatever fits your delivery rhythm
- **Event-driven:** architecture change, new risks identified, regulatory shift, major team change
- **Early-warning signals:** spike in escaped defects, increased customer complaints, delivery slowdowns, performance degradation

Leading vs lagging indicators

Leading indicators give you early signals to adjust before failures hit production:

- **Change failure rate:** percentage of changes causing production failures
- **Mean time to detect:** how quickly you spot a defect after it is introduced
- **Test reliability:** flaky test rate (if tests are noisy, evidence quality is low)
- **Static analysis trend:** are critical findings increasing or decreasing?

Lagging indicators confirm whether risk actually went down:

- **Escaped defects** (severity-weighted)**:** defects found in production
- **Production incidents:** count, severity, user impact
- **Customer churn attributed to quality**
- **External failure cost:** the realized cost of production failures

Use leading indicators to steer. Use lagging indicators to validate. Neither alone tells the full story.

CoQ tracking over time

Track all four CoQ categories over time. The pattern you want to see: external failure costs (especially high-severity incidents) trending down, while prevention and appraisal spend becomes deliberate and stable relative to delivery volume.

Two useful ratios:

- **Shift-left ratio:** (Prevention + Appraisal) / (Internal failure + External failure). Should increase over time.
- **External failure ratio:** External failure / Total CoQ. Should decrease over time.

In early or maturing quality systems, external failure costs often dominate. The economic goal is to reallocate toward prevention and appraisal to reduce failure costs. In mature systems, failure costs are lower and investment is tuned and justified.

The rebalancing decision framework

For each testing approach in your portfolio, the review should produce one of five decisions:

1. **Increase:** evidence shows strong risk reduction per unit cost
2. **Reduce:** weak outcomes, high cost, or misaligned with prioritized risks
3. **Stop:** little evidence produced (redundant, correlated, or obsolete)
4. **Add:** new risks, coverage gaps, or failure modes not addressed
5. **Accept residual risk:** document rationale, assign owner, set review date

Every increase should name what gets reduced or stopped. Budgets are finite.

Running a review meeting

A structured review meeting turns measurement into decisions. Collect data beforehand: effectiveness and efficiency metrics, the risk register, defect and incident summaries. The meeting itself covers four questions:

1. **Are prioritized risks trending down?** Look at escaped defects and incident severity for your top risks.
2. **Are we getting credible evidence at acceptable cost?** Look at effort per risk, test reliability, and feedback speed.
3. **What changed?** New risks, architecture shifts, changed business priorities, team changes.
4. **What do we rebalance?** For each finding, decide: increase, reduce, stop, add, or accept.

Timebox debates. Use data to anchor the discussion. Force decisions. Capture action items with an owner, a due date, and a success metric. A review that produces no decisions is a status meeting, not a review.

Running example continued: Six months in, the review for "payment processing failures due to regression defects" shows:

- Payment incidents dropped from 4 per quarter to 1. External failure cost dropped from ~$200K to ~$50K per quarter.
- Unit test suite catches most logic errors early. Contract tests caught two API changes that would have caused silent failures.
- The system-level E2E test is flaky and slow, but the one incident that slipped through was a system-scope configuration issue that only the E2E test could have caught.

Decision: keep unit and contract tests (high return), accept the flaky E2E test for now but invest in stabilizing it (the risk it covers is real), and add a design review gate for payment module changes (prevention is cheaper than any of the above). Reduce the manual regression cycle for payment flows (now covered by automation).

The feedback loop

The four-step process is a cycle, not a sequence. Reviews may surface new risks that were not identified in Step 1, or changes in the risk landscape that require revisiting priorities in Step 2. When that happens, loop back to the appropriate step:

- **New risks discovered:** return to Step 1
- **Priorities changed** (business shift, lifecycle change)**:** return to Step 2
- **Portfolio underperforming** (poor risk reduction, unacceptable escapes)**:** return to Step 3

KEY TAKEAWAY

Review your testing portfolio on a regular cadence using both leading and lagging indicators. For each approach, decide: increase, reduce, stop, add, or accept. Track CoQ over time to see whether you are shifting spending from expensive failure costs toward cheaper prevention and appraisal.

The per-risk CoQ comparison

This is the most practical tool in the framework. For each risk in your portfolio, CoQ lets you lay three options side by side:

1. **Cost to keep absorbing failures** (do nothing new)**:** what are you currently spending on internal and external failure for this risk class? Incident response, customer support, hotfixes, lost revenue, rework. Pull from your own data: incident reports, support tickets, engineering time logs.
2. **Cost to catch earlier through appraisal** (testing and reviews)**:** what would it cost to add or improve testing that targets this risk? Automated checks, exploratory sessions, code reviews, static analysis.
3. **Cost to prevent** (prevention practices)**:** what would it cost to reduce the likelihood of this risk through prevention? Design reviews, training, coding standards, threat modeling, architecture improvements.

Failure costs are bigger than engineering thinks

Engineering only sees part of the picture: debugging time, rework, hotfixes. But failures cost the whole organization. Sales knows about deals lost to quality reputation. Marketing knows about campaigns disrupted by outages. Customer success knows the real churn and support ticket numbers. Infrastructure knows incident response costs, overtime, emergency scaling.

When you build the failure cost estimate, talk to these departments. Their data will almost certainly show that failure costs are higher than engineering alone would estimate. This strengthens the case for prevention and appraisal. And it has a second effect: when other departments contribute their failure cost data, they become stakeholders in the quality investment decision. Quality stops being ‘a QA concern’ and becomes an organizational cost everyone can see.

Worked example: payment processing

Risk class: “Payment processing failures due to regression defects”.

Option 1 - Absorbing failures:

From incident data, the team had 4 payment incidents last quarter. Average cost per incident (engineering response + customer support + lost transactions + credits issued): optimistic $30K, most likely $50K, pessimistic $80K.

Per quarter: optimistic $120K, most likely **$200K**, pessimistic $320K.

Option 2 - Catching earlier through appraisal:

Add automated regression suite for payment flows at unit and contract levels, plus monthly exploratory sessions. Setup cost: $40K. Ongoing per quarter: optimistic $15K, most likely **$20K**, pessimistic $30K. Expected to catch ~70% of the defect class before release.

Option 3 - Preventing:

Add mandatory design reviews for all payment module changes, plus pair programming for critical paths. Per quarter: optimistic $10K, most likely **$15K**, pessimistic $25K. Expected to reduce defect introduction by ~50%.

The comparison: absorbing costs $200K per quarter. Catching earlier costs $20K per quarter (after setup). Preventing costs $15K per quarter. The cheapest credible mix is probably some prevention plus some appraisal, and the numbers make the case without needing to ‘prove’ that prevention works in the abstract.

This does not prove prevention. It makes the choice explicit. The comparison lets the decision-maker see the trade-off and choose deliberately, rather than defaulting to ‘ship and fix later’ because the alternative was invisible.

‘Cheapest credible’ is the key phrase. Not the cheapest option in theory, but the cheapest combination that actually works: that produces credible evidence the risk is being managed. Pure prevention alone rarely eliminates a risk. Pure appraisal alone catches defects but does not reduce their introduction. The optimal mix for most risks is some prevention (to reduce defect introduction) plus some appraisal (to catch what slips through) plus operational controls (to limit blast radius when something still gets to production).

When management asks ‘why should we fund this?’, you point at the risk, the CoQ comparison, and the agreed threshold. When an incident happens, you point at the documented risk acceptance decision and the owner who signed it. Quality stops being a debate and becomes a documented, reviewable investment decision.

KEY TAKEAWAY

For each major risk, compare the cost of absorbing failures, catching them earlier, and preventing them. Talk to departments beyond engineering for the full failure cost picture. The comparison makes the trade-off visible and turns quality investment into a shared organizational decision.

How to start Monday morning

You do not need to overhaul your testing strategy in one go. Start small.

1. **Pick one painful failure class.** Look at your recent incidents. Which one cost the most in engineering time, customer impact, and organizational disruption? Pick that one.
2. **Write an explicit risk statement.** “If [condition], then [consequence], resulting in [impact]”. Make the trigger observable, the consequence measurable, and the owner identified.
3. **Build a small CoQ baseline.** Use existing data: incident costs, mitigation costs, testing costs, support costs. Talk to sales, marketing, customer success, infrastructure. You do not need perfect data. Rough estimates from the people who lived through the incidents are enough to start.
4. **Compare the three options for that risk.** Cost to keep absorbing failures. Cost to catch earlier through appraisal. Cost to prevent. Lay them side by side. Propose the cheapest credible mix.
5. **Agree on review signals.** What will you look at in 3 months to know if the mix is working? Pick one leading indicator (detection speed or change failure rate) and one lagging indicator (escaped defects or incident cost for that risk class). Set the review date now.

That is it. One risk. One CoQ comparison. One proposal. One review cycle.

If it works, expand to the next failure class. Build the portfolio gradually. Each cycle that reduces costs builds the case for the next one. Your authority as a quality professional grows with demonstrated results, not with assertions about best practices.

The portfolio grows from evidence, not from a big-bang transformation. And the conversation with management shifts from ‘trust me, we need more testing’ to ‘here is what the last investment saved, and here is the next highest-return opportunity’.

KEY TAKEAWAY

Start with one painful failure class, one CoQ comparison, one proposal. Prove the approach works, then expand. The portfolio grows from evidence.