The 1–10–100–1,000 Rule in Software Engineering

Henrik Andersen
M.Sc. in Electrical Engineering

Abstract

The 1–10–100–1,000 rule expresses a planning intuition: a defect can become more expensive to resolve as its effects spread into dependent code, integrated systems and deployed products. This paper examines the limits of fixed lifecycle multipliers and describes how requirements review, failure modes and effects analysis, and targeted verification can reduce exposure to late discovery. A worked embedded-software example shows how to estimate the cost of correcting the same defect at different discovery points using explicit assumptions. The example is hypothetical and the paper reports no original measurements.

1. What the rule represents

The rule assigns illustrative relative costs of 1, 10, 100 and 1,000 to defects corrected at progressively later points. The four points used here are design, implementation, integration test and field operation. Use the sequence to ask what additional work a defect would create if it passed a given checkpoint.

Discovery point	Illustrative index	Potential scope of correction
Design	1	Revise a requirement or interface and review the change.
Implementation	10	Diagnose, change code and update unit tests.
Integration test	100	Reproduce the defect across components and repeat affected system tests.
Field operation	1,000	Assess affected installations, deploy a fix, support customers and verify recovery.

Table 1. Illustrative lifecycle cost indices. Release and test obligations depend on the product.

2. What the evidence supports

A study of 171 Team Software Process projects found little evidence of large, consistent increases in pre-delivery issue-resolution effort with later discovery. Its dataset did not cover post-delivery deployment (Menzies et al., 2017). Corrective engineering effort, elapsed time and field impact require separate measurement. Estimate them from the product’s actual dependencies, release process and exposure.

3. An evidence-led engineering workflow

3.1 Identify consequential failure modes

Start with the function, its operating conditions and the consequence of losing it. In a software design FMEA, examine architecture, interfaces, state transitions and fault responses. In a process FMEA, examine how activities such as configuration, test execution or deployment can fail. IEC 60812 applies FMEA to software, processes and interfaces as well as hardware (IEC, 2018).

Record each failure mode, its local and system effects, plausible causes, existing controls and any action required. Assign an owner and define the evidence that will close the action. A high-consequence failure warrants attention even when historical cost data are sparse.

3.2 Put checks where they can discriminate

Make requirements and interface assumptions testable at design review. Use review, static analysis and unit tests for behaviours that can be checked locally. Use integration and system tests for failures that depend on sequencing, hardware, timing or operating conditions. For firmware, hardware-in-the-loop testing can exercise restart and communication faults on representative hardware. Security-focused source review and executable testing are also recommended in the Secure Software Development Framework (Souppaya, Scarfone and Dodson, 2022).

3.3 Estimate and record the cost of correction

For a given defect, define the scope before comparing discovery points. Estimate direct cost as engineering hours multiplied by a stated labour rate, plus incremental external costs such as field service and distribution. Record incident handling, support, downtime and commercial impact separately when they fall outside that direct cost boundary. Apply the same cost boundary to every scenario and disclose any excluded items.

For actual defect records, retain the suspected introduction point, discovery point, defect class, affected baseline, correction effort and deployment effort. Compare like defects where possible. Differences in severity, architecture or release obligations can dominate a comparison based on discovery stage alone.

4. Worked example: stale motor-enable command

An embedded controller retains a motor-enable command across a communication timeout. On reconnection it can reuse the stale command before receiving a fresh one. The design requirement is to enter a defined safe state after timeout and, once communication returns, require a new enable command before energising the output.

A state-transition review can expose the missing requirement before code is written. A unit test can check command invalidation. An integration test can interrupt and restore communication on representative hardware and check the physical enable output. If the defect reaches the field, the same change also requires identifying affected versions, distributing the update and verifying installation.

4.1 Compare four discovery scenarios

Assume a blended labour rate of €100 per hour. The four scenarios are alternative discovery points for the same defect. The estimates include incremental work from discovery through verification at that point. The field scenario also includes rollout and an assumed €2,000 in external field-service costs.

Discovery point	Included work	Effort	Direct cost	Ratio
Design	Specify timeout and re-enable behaviour, then review the state model.	2 h	€200	1×
Implementation	Correct state handling and run targeted unit tests.	8 h	€800	4×
Integration test	Reproduce on hardware, correct the code and rerun affected integration tests.	24 h	€2,400	12×
Field operation	Triage, patch, qualify and roll out the update. Includes €2,000 external field service.	80 h	€10,000	50×

Table 2. Hypothetical direct correction costs. Ratios are relative to the €200 design scenario.

Verify the change by inducing a timeout with an active enable command, restoring communication without a new command and checking that the output remains de-energised. Then send a fresh valid command and confirm recovery. Repeat on affected configurations. The cost estimate excludes downtime, lost revenue and any consequence of unintended motion.

5. Conclusion

Use the 1–10–100–1,000 sequence to identify where a defect could acquire additional correction and deployment costs. Base investment in earlier detection on failure consequences, the checks available at each stage and measured effort from comparable defects.

References

International Electrotechnical Commission (IEC) (2018) *Failure modes and effects analysis (FMEA and FMECA)*. IEC 60812:2018. IEC publication record.

Menzies, T., Nichols, W., Shull, F. and Layman, L. (2017) ‘Are delayed issues harder to resolve? Revisiting cost-to-fix of defects throughout the lifecycle’, *Empirical Software Engineering*, 22(4), pp. 1903–1935. doi:10.1007/s10664-016-9469-x.

Souppaya, M., Scarfone, K. and Dodson, D. (2022) *Secure Software Development Framework (SSDF) Version 1.1*. NIST SP 800-218. doi:10.6028/NIST.SP.800-218.