ScaleQuality Code Maturity Model (SQCM)

An evidence-based code maturity model for the era of AI-generated software

Version 2.0 · August 2026

Author: Erik Fernandes · Founder, ScaleQuality

Contact: erik.fernandes@scalequality.io · scalequality.io

Supersedes: SQCM v1.0, July 2026 · DOI 10.5281/zenodo.21251863

DOI: *(assigned on publication)*

Executive summary

Version 1.0 of this model defined **code maturity** as a measurable construct at the repository level, assessed exclusively from observable evidence, with no questionnaires and no self-declaration. It evaluated four domains: Security, Reliability, Maintainability, and a fourth not represented as a first-class domain in the standards reviewed in v1.0, **AI code durability**.

Version 2.0 keeps that construct unchanged and extends it along one axis and one principle.

The axis is a **fifth domain: Supply Chain**. A repository's maturity cannot be read from the code its team wrote alone. A substantial portion of what an application ships was written by someone else and pulled in by a manifest, a base image, or an infrastructure template. A model that measures only first-party code measures only part of the shipped artifact.

The principle is what we call the **measurement boundary**: *measure each dimension where its artifact actually exists*. Some properties are legible in the repository. Others exist only in the built container image, and can therefore only be measured in the pipeline that built it. v2.0 makes that boundary explicit rather than pretending a repository scan can see everything, and it specifies where each dimension is measured and what is reported when the artifact is out of reach.

v2.0 also reports, for the first time, **primary measurements of the model's own instrument**: the analysis completes in about five seconds in a continuous-integration gate, and this paper publishes the per-stage breakdown behind that number, along with two methodological failures we found while measuring it. A model that asks organizations to trust measurement owes the same discipline to the measurement of itself.

---

1. What v1.0 established, and what it left open

Version 1.0 argued that the existing rulers answer questions adjacent to the one teams actually ask. ISO/IEC 25010 and ISO/IEC 5055 measure present quality [1][2]. CMMI measures organizational

process [6]. DORA measures delivery flow [7][21][22]. None answers *"can I trust this code?"* about a specific repository, today, without a consulting engagement.

SQCM v1.0's answer had four load-bearing commitments, all preserved in v2.0:

1. **Evidence, never declaration.** Nothing self-reported enters the score.
2. **Explicit confidence, and unmeasured states that do not score.** Absence of evidence is reported as absence, never interpolated into a neutral number.
3. **Zero configuration.** A repository assessment starts from a pointer and finishes in minutes, because an assessment that takes weeks measures a repository that has already changed. Artifact-scoped evidence (section 3) is additive and collected only where the artifact exists; it does not reintroduce a configuration step for the repository assessment.
4. **Reproducibility through score versioning.** Past verdicts are immutable; the model's evolution applies forward.

Three limitations recorded in v1.0 section 8 are what v2.0 addresses:

- The model read the team's own code and its practices, but not **what the repository imports and ships**.
- Static analysis was **intraprocedural**: it could see a dangerous pattern in one function, but not a dangerous *path* across files.
- The verdict covered application source, but not the **infrastructure and packaging** that determine how that source is deployed.

---

2. The fifth domain: Supply Chain

2.1 Why it is a domain and not a finding category

In most modern applications a large share of shipped bytes were not written by the shipping team. They arrive through a package manifest, a transitive dependency of a transitive dependency, a base image, or an infrastructure template. Two properties make this a distinct maturity domain rather than a subcategory of Security:

It has a different failure mode. First-party security weaknesses are authored: someone wrote the vulnerable pattern. Supply-chain exposure is *inherited*: the code was correct when adopted and became vulnerable while nobody touched it. A repository can degrade in this domain with zero commits. No other domain has that property, and it is precisely why point-in-time assessment is insufficient and continuous measurement matters.

It has a different remediation. A first-party weakness is fixed by changing the code. Supply-chain exposure is usually fixed by changing a version constraint, which is a different decision, taken by different people, with different risk (a version bump can break the build, which is why upgrade discipline is itself a maturity signal).

The industry has converged on this separation independently: the provenance and integrity frameworks that emerged after the high-profile build-system and package compromises of the last several years treat the supply chain as its own layer of assurance, with dedicated controls and checks

and, in the case of the provenance framework, progressive assurance levels [28][29]. Regulatory movement points the same way, with component inventory and vulnerability handling becoming regulatory obligations in defined jurisdictions and product scopes rather than good practice alone [30] [31].

2.2 What the domain measures

Supply Chain in SQCM v2.0 aggregates three families of observable evidence:

- **Known-vulnerable components.** Declared and transitively resolved dependencies matched against public advisory data [32]. The signal is the presence of components with published advisories, weighted by advisory severity and deduplicated by advisory identifier, so that one advisory affecting eight packages is one problem to fix, not eight.
- **Shipped image contents.** The operating-system and runtime packages present in the container image the pipeline actually builds, including whether the base distribution still receives security updates at all (section 3 explains why this is measured elsewhere).
- **Licence obligations.** Strong-copyleft and unresolvable licences among the dependencies.

The third family is **reported and never scored**, and the reason is a design principle worth stating: *"is AGPL acceptable here?"* is a company policy question, not a code defect. A scanner that answers it on the company's behalf is making a legal decision it has no standing to make. SQCM surfaces the obligation and stops.

2.3 What the domain deliberately does not claim

Presence of an advisory is not proof of exploitability. A vulnerable function that is never called from any reachable entry point is a real obligation (it will be flagged by every auditor and every customer questionnaire) but not a real exposure. SQCM v2.0 reports **presence**, labelled as presence, and does not claim reachability. Section 7 records reachability analysis as declared future work rather than an implied capability.

---

3. The measurement boundary

3.1 The principle

Some properties of a system are legible in its repository. Others are not legible anywhere except in a built artifact that the repository merely describes.

Version 1.0 assumed, implicitly, that the repository was the unit of measurement. Version 2.0 replaces that assumption with an explicit rule:

> **Measure each dimension where its artifact exists. Where the artifact is out of reach, report the dimension as unmeasured. Never substitute a proxy and present it as the thing.**

This sounds obvious and is routinely violated. The concrete case that forced us to state it is container image vulnerabilities.

3.2 The container image case

A container image's vulnerabilities live in the image: its operating-system packages, its runtime, the layers a build produced. A repository contains a *recipe* for that image, not the image.

Three approaches are available, and two of them are dishonest:

1. **Scan the base image referenced in the Dockerfile.** Cheap to implement and wrong: the base is not what the customer deploys. Everything the build adds, and every layer a private registry contributes, is invisible. It also costs a full image pull per analysis, which destroys the assessment's latency property (section 5).
2. **Infer from the tag.** Report the base image as stale or end-of-life based on its name. This is a heuristic on a string, presented as a measurement of an artifact.
3. **Measure the image where it exists.** The pipeline that just built the image is the only place the image is present, local, and free to inspect. Scan it there, and transmit only the result.

SQCM v2.0 takes the third path. The consequence is a property most measurement instruments do not have: **a dimension can be measured or not depending on where the measurement is invoked**, and the verdict says which. A repository assessment reports image contents as unmeasured. A pipeline assessment with the image reference supplied reports them as measured, with the image identified in the finding itself.

Two disciplines make this safe rather than confusing:

- **The unmeasured state is never a clean state.** An assessment that did not receive an image report says so. It does not report zero image vulnerabilities.
- **Withholding evidence cannot improve the verdict.** Supplying the image report can only lower a score or leave it unchanged, never raise it, so an organization cannot game the verdict by withholding the image. The property we guarantee is this directional one, stated deliberately rather than the stronger and harder claim that every possible added measurement is monotone.

3.3 The same boundary, elsewhere

Once stated, the principle applies beyond containers. Infrastructure-as-code templates *are* in the repository, so misconfiguration is measured there. Runtime behaviour is not in the repository and is not measured at all, which is why v1.0 excluded Performance Efficiency [2] and v2.0 continues to. The boundary is what distinguishes a deliberate scope from an accidental blind spot.

---

4. Signal families added in v2.0

Beyond the fifth domain, v2.0 adds three signal families to existing domains. Each is stated with what it measures and what it deliberately refuses to measure.

Infrastructure and container misconfiguration (into Security). Container definitions, compose files, infrastructure templates, orchestration manifests and cloud-formation templates are evaluated against a published misconfiguration catalogue: containers running as root, storage without encryption, network rules wider than they need to be, transport without TLS.

Two calibration decisions here are, we believe, the difference between a signal and noise. First, informational-severity checks are dropped: on a real infrastructure repository they were the majority of all findings and none of them were decisions anyone would act on. Second, a small number of checks that describe **architectural intent rather than a defect** are deliberately demoted. A public product is *supposed* to have an internet-facing load balancer. A verdict whose top-ranked risk is "your public application has a public load balancer" is a verdict that trains its reader to stop reading, and a model that trains people to ignore it has failed regardless of its recall.

Related, and stated as a principle: infrastructure misconfiguration **does not trigger the critical-exposure cap** that a leaked credential or a shipped critical advisory triggers. A misconfiguration is a posture judgement against a baseline; a versioned credential is proof of exposure. Treating them as equivalent would make the strongest signal in the model indistinguishable from its weakest.

Code duplication (into Maintainability). Duplication is the defect no linter reports: the code is correct, it is simply written four times, so every fix has to be found four times. It is also, per the industry evidence assembled in v1.0 section 3.3, a documented signature of AI-assisted development [17][18][19] — which makes it a direct read on the durability thesis rather than a borrowed metric.

The model's clone threshold is aligned with the thresholds established clone detectors use (on the order of ten successive lines and one hundred tokens; exact thresholds vary by tool and language), so the reported percentage is comparable to a number practitioners already know rather than identical to any one tool's. Two calibration decisions again matter more than the detection:

- **Below a threshold, duplication does not move the score at all.** Some repetition is the honest cost of clarity, and an instrument that punishes all of it pushes teams toward premature abstraction, which is a worse outcome than the duplication was.
- **Only source code is measured.** This is not obvious and cost us a false result before we caught it. Run without restriction, the detector reported **8.5% duplication** on one of our own repositories. Restricted to source formats, the same repository measured **1.7%**. The difference was translation catalogues, generated bundles, and lockfiles — files that are *supposed* to be repetitive. The first number was not a measurement of anything.

Container image vulnerabilities (into Supply Chain), per section 3.

---

5. Speed as a property of the model

5.1 Why latency belongs in a methodology paper

Version 1.0 argued that zero configuration is an epistemological requirement, not a convenience: an assessment that requires weeks measures a repository that has already changed. Latency is the same argument at a shorter timescale.

If maturity is to function as an operational contract — a line in the delivery pipeline below which the build fails — then the assessment must fit inside the pipeline. An instrument that adds minutes to every pull request will be disabled, and a disabled instrument measures nothing. The speed of the measurement is therefore a property of the model, not a detail of its implementation.

5.2 Measured

On a mid-size repository, the complete assessment inside a continuous-integration gate measured **5.25 s, 5.29 s and 7.17 s** across three consecutive runs, end to end from upload to verdict. Measurement conditions, stated so the number can be attacked: a two-vCPU / eight-GB container on the staging cluster, the fast engine with all stages present, an adaptive poll interval (see 5.3), and three consecutive runs on the same commit. Three runs is a small sample and is reported as such; a larger campaign with median and tail latency across a range of repository sizes is declared work in section 7, not claimed here. Stage timings, all executing concurrently:

STAGE	MEASURED
Static analysis (security, reliability, maintainability)	5.12 to 5.21 s
Test-coverage read	2.03 to 2.50 s
Dependency advisories	1.08 to 1.33 s
Duplication	1.05 to 1.16 s
Secret detection	0.12 to 0.15 s
Infrastructure presence check	25 to 54 ms *(scanner skipped, no infrastructure present; the scanner itself costs ~1.2 s warm, see 5.3)*

Because the stages run concurrently, the assessment costs approximately the longest stage rather than their sum.

The number that matters for the model's claim is not the absolute figure but its relationship to the change cycle: **the assessment fits inside a pull request without being noticed.**

5.3 Two methodological failures, published deliberately

A paper that publishes an instrument's speed should publish how the measurement of that speed went wrong, because both failures are general and both would have produced a number we would have believed.

Failure one: a cost we attributed to the wrong thing, twice. Adding the infrastructure misconfiguration scanner moved the assessment from 4.4 s to 7.1 s on the same repository. The obvious reading was that infrastructure scanning is expensive.

The second reading was better but still wrong. The scanner's cost was *independent of repository size* — 3.47 s on a repository of a few dozen files, 3.31 s on a much larger one — so we concluded it paid a large fixed cost compiling its policy bundle before reading a single file. That conclusion produced a correct fix for a partly incorrect reason: the scanner is now invoked only when infrastructure files exist,

established by a filesystem check costing about a millisecond, and repositories without infrastructure pay nothing.

Measured properly in isolation, the scanner takes about **1.2 s** warm, and five concurrent invocations complete in **about 2 s in total** on a machine with cores to spare. The size-independence was real; the magnitude was not. Most of the 3.4 s was **contention** with the two other CPU-bound stages on a two-core machine, not work the scanner was doing.

The general lesson is the one this section is about: on a saturated machine, every stage's measured duration includes the other stages, and a per-stage timing table reads like an attribution when it is partly a division of scarcity. The fix survived the correction — skipping a scanner that has nothing to scan is right regardless — but the number we would have published as "the scanner costs 3.4 s" was a property of the machine, not of the scanner.

Failure two: an artefact of the measuring instrument reported as a property of the system. The pipeline client polled for the verdict at a fixed five-second interval. The assessment completes at approximately five seconds. The consequence is that the *same repository at the same commit* measured **5.26 s or 10.28 s** depending on which side of a poll boundary the analysis happened to land. Neither number was wrong and neither was a measurement of the system: the five-second difference was the instrument waiting, not the system working. With an adaptive interval, three consecutive runs returned 5.25 s, 5.29 s and 7.17 s.

The general lesson is not about polling. It is that **a measured number inherits the resolution of whatever measured it**, and an instrument reporting at coarser granularity than the phenomenon will produce stable, reproducible, entirely artefactual results. We publish this because we nearly quoted the 10.28 s figure as our own benchmark.

---

6. Interprocedural analysis

6.1 The limitation this addresses

Pattern-based static analysis matches shapes in a single file. It sees `eval(userInput)` when both appear together, and it does not see the case that actually occurs in production code, where the untrusted value enters through one function, is passed through two others, and reaches the dangerous operation in a fourth file. That path is where real exploitable defects live, and it is invisible to a pattern matcher by construction.

Closing it requires **taint analysis**: building a graph of the program and following values from untrusted sources to sensitive sinks across function and file boundaries.

6.2 Why we built our own

Interprocedural analysis is available commercially, and the licensing terms of the leading options were the deciding factor rather than the technology. One category is proprietary and would require an OEM arrangement to resell as part of a product. Another category is available at no cost but under terms that restrict use outside open-source and research contexts, including automated analysis of others' code as

a hosted service [34], which is precisely what a product does. A capability we cannot legally build a product on is not a capability.

SQCM v2.0's deep analysis is therefore built on a permissively licensed (Apache 2.0) open-source code-property-graph platform [33], with a taint specification authored by us. The specification is model property and is not published here, on the same terms v1.0 set for the scoring weights.

6.3 What was measured during calibration

The result worth publishing is not the rule set but the **calibration behaviour**, because it quantifies a trade-off every static analysis product faces and few report.

Against a deliberately vulnerable reference application, successive refinements of the specification produced **137, then 25, then 4, then 0 false positives**, while a fixed regression suite of known-true paths stayed at **5 of 5 detected** throughout.

What the arc establishes is a design position: the specification was tuned toward precision even at the cost of recall, on the reasoning that **a false positive costs more than a missed finding** in an instrument whose entire value is being believed. A tool that cries wolf is switched off, and a tool that is switched off has recall zero.

6.4 Extending the specification, and what the extension cost

The initial specification covered five vulnerability classes in one language. It now covers **ten classes in Java and seven in JavaScript/TypeScript**, each carrying its CWE identifier on the finding itself rather than in a coverage table. The two catalogues are deliberately asymmetric: unsafe deserialisation and reflection by name are Java problems that barely exist in Node, and forcing a rule across a language where its shape does not occur produces noise, not coverage.

The extension is reported here because of what it cost. Running the new rules against real code produced **ten erroneous findings: nine false positives and one misclassification**, and all ten shared a single root cause: **a sink identified by method name without a constraint on its receiver**. `getInputStream` also exists on a process, an archive and a classpath resource; `setHeader` is also a JSON Web Token builder method; `execute` is also an ordinary domain method on an ordinary object. Each was corrected by requiring the receiver to be what the rule means, and each correction was verified against the corpus rather than assumed.

The one misclassification among those ten deserves separate statement, because it is a different kind of error from the other nine. A rule reported a genuine tainted path under the wrong class: data read from an outbound HTTP response reached a write, and the rule called it file access. The flow was real; the classification was not. **Assigning the wrong class is not a lesser error than a false positive** — it is the error that causes a security team to stop reading the report, because it demonstrates the instrument does not understand what it is looking at.

Validation now runs against a labelled corpus rather than a single application:

SUBJECT	RESULT
Deliberately vulnerable Java application, 277 source files	23 findings, **each in the package named for its own vulnerability class**, 23 of 23
Deliberately vulnerable Node application	5 findings, all genuine, including one the application's own source comments describe as insecure
Reference Java application in ordinary health, 49 files	**0 findings**, which is the correct result
ScaleQuality's own production code: six services and a web front end	**0 false positives**

The corpus is stronger than the single-application arc it replaces, because in the two vulnerable subjects the directory structure encodes the expected class, which gives an independent check on classification and not merely on detection. It remains short of the general precision and recall claim declared as future work in section 7: these are public teaching applications, whose defects are written to be found.

6.5 Two failures of the environment, not of the rules

Section 5.3 published two ways a measurement of speed can be an artefact of its instrument. Extending the deep engine produced two of the same species, and both are properties of *where the analysis runs* rather than of what it analyses. Both are reported because both are invisible by construction.

The engine can fail to run at all, silently. The graph platform reads its own query file using the host's default character encoding. In a container with no locale configured, that default is ASCII, and a single accented byte in a comment aborts the query. The parse stage succeeds; only the query dies; and a failed query is correctly recorded as *not measured*. The result is an analysis that never happened, reported as an absence of analysis rather than as an error — which is the honest outcome, and also the reason nobody noticed. The condition had been latent since the first non-ASCII comment was written.

A rule can disappear in one environment and not another. Type resolution for chained calls depends on the runtime executing the parser. The same construct, the same engine version and the same source resolved to a fully qualified type on one machine and remained unresolved on another. Because the precision constraints described in 6.4 are expressed in terms of resolved types, the affected rule found nothing where the type did not resolve: ten findings on one machine, nine on the other, with no error on either. Receiver constraints are now satisfied by the resolved type **or** by the text of the call, so a rule degrades in precision rather than vanishing.

The general lesson is the one that governs section 5.3 as well, at a different layer: **an instrument's results are a property of the instrument and its environment together**, and a validation run on the author's machine measures a system nobody else will run. Validation is now performed inside the published artifact, and the two failures above are exactly what that discipline caught.

6.6 Why it does not run on every assessment

Graph construction requires its own runtime and gigabytes of memory. Measured inside the published artifact against the 277-file Java subject, the complete deep cycle costs **about 33 seconds** on eight

virtual CPUs, against **67 seconds** on four and **101 seconds** on two; the near-linear scaling is a property of a parser that parallelises per file. Running that inside the five-second gate is not possible, and pretending otherwise would sacrifice section 5's property to gain section 6's.

The scaling has a second-order consequence worth recording, because it inverts the usual reasoning about compute: since the platform bills by the second, the eight-CPU configuration costs roughly three times as much per second and finishes roughly three times sooner, so **cost per assessment is approximately unchanged while the wait falls to a third**. For an out-of-band analysis someone is waiting for, that is the difference between a result and an interruption. The two are therefore **two products of the model, not two settings**: the fast assessment runs on every change, and the deep assessment runs out of band, on its own schedule, with its result attached to the repository when ready. Any assessment that requests deep analysis where the deep engine is unavailable is served by the fast engine **and records that it was**, so the provenance never claims a depth it did not deliver.

---

7. Limitations and validation agenda

Version 1.0 declared six limitations. Four remain unchanged and are not repeated: observable evidence has blind spots; performance is out of scope; AI-authorship attribution is heuristic; correlation is not causation.

Two v1.0 items are updated, and four are new to v2.0.

Updated — the level thresholds remain design calibration. The bands are still derived from the five-level tradition [6] and design experience, not from a longitudinal outcome study. The agenda is unchanged: correlate levels against observable outcomes in a continuously measured cohort, publish level distributions for public calibration, and revise with an incremented score version that preserves history.

Updated — third-party evidence for the durability domain retains its known biases [17][18][19][26][27], with the same qualifications v1.0 recorded.

Corrected during v2.0's own development — Supply Chain saturation. The domain originally composed by subtracting a fixed deduction per distinct advisory. Because advisory counts on real repositories are large, it reached its floor almost immediately: three critical advisories, or four high, or ten moderate, and the score was zero. Every real subject we measured scored zero, which is a true statement that has stopped discriminating: it could not distinguish a repository with a manageable backlog from one with an unmanageable one, and that distinction is the only thing the reader needs.

The composition is now exponential decay over weighted, deduplicated advisory units. One critical advisory still lands in the critical band, severity remains monotone, and more advisories is never better; what changed is that the range where a team can actually act now spreads across the scale instead of collapsing onto its floor. The change shipped as a score version increment under the reproducibility discipline of v1.0 section 6: prior verdicts retain the version that produced them and are never silently recompared against the new curve.

We record this as a limitation rather than a feature because it is one: the original calibration was published in v1.0's spirit and was wrong in a way only measurement revealed, and the same risk applies to every threshold in this model that has not yet met a large enough population. That is the agenda in the item above.

New — presence is reported, reachability is not. Section 2.3. Reachability analysis is a declared roadmap item and is deliberately not implied by any current output.

New — the deep analysis specification is validated against a labelled corpus, not against a general one. Sections 6.3 and 6.4. Detection and classification are now checked against public applications whose directory structure encodes the expected vulnerability class, and against production code for false positives, across two languages. What that establishes is behaviour on defects written to be found. General precision and recall remain unestablished, and the agenda is an independent corpus not authored as teaching material, plus language coverage beyond the two.

New — the speed benchmark is a small sample, not a campaign. Section 5.2. The reported figures are three runs on one repository under stated conditions. A defensible latency claim needs median and tail across a range of sizes and languages, on pinned instrument and hardware versions; that campaign is agenda, and the current numbers are offered as an order-of-magnitude property, not a benchmark result.

New — the measurement boundary shifts what is comparable. Because some dimensions are measured only where their artifact exists (section 3), two assessments of the same repository can legitimately cover different scope. The model addresses this with the directional guarantee of section 3.2 (withholding evidence cannot improve the verdict) and with per-scanner provenance recorded on every verdict, so that any two verdicts can be compared knowing exactly what each one saw. Users comparing scores across differently instrumented pipelines should read the provenance, and the product surfaces it for that reason.

---

8. Conclusion

Version 1.0 argued that the AI era needs a ruler for a question the existing rulers do not answer, and defined code maturity as that ruler: measured from observable evidence, without declaration, in minutes, including a durability dimension not represented as a first-class domain in the standards reviewed in v1.0.

Version 2.0 makes two corrections to that instrument, and both came from measuring rather than designing.

The first is that a repository is not the whole artifact. A substantial part of what an application ships can arrive from somewhere else, so a maturity model that reads only first-party code evaluates only part of the delivered artifact. Supply Chain enters as a first-class domain for that reason.

The second is that the repository is not the only place measurement happens. Some properties exist only in a built artifact, and the honest response is to measure them where they exist and to declare them unmeasured where they are out of reach — never to substitute a proxy and present it as the thing. That principle, more than any individual scanner, is what version 2.0 adds to the model.

Both corrections point the same direction as version 1.0's founding commitment. An instrument earns the right to be believed by being explicit about what it saw, what it did not, and how confident it is in the difference. Version 2.0 extends that discipline to the instrument's own measurements of itself, including four occasions on which a number was an artefact of how or where we were measuring rather than a property of what we measured: twice in timing the fast assessment, and twice in the environment the deep assessment runs in, where the failure mode was not a wrong number but a silent absence of one.

The model is in operation at scalequality.io. The methodology remains published, versioned, and open to scrutiny, because that is what separates a ruler from an opinion.

References

References [1] through [27] are carried from SQCM v1.0. They are reproduced here in full so this document resolves its own citations without requiring the v1.0 record.

[1] ISO/IEC 25010:2023. *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model*. International Organization for Standardization, 2023. https://www.iso.org/standard/78176.html

[2] ISO/IEC 5055:2021. *Information technology — Software measurement — Software quality measurement — Automated source code quality measures*. International Organization for Standardization, 2021 (developed by CISQ/OMG). https://www.iso.org/standard/80623.html · https://www.it-cisq.org/standards/code-quality-standards/

[3] Letouzey, J.-L. "The SQALE method for evaluating Technical Debt". *Third International Workshop on Managing Technical Debt (MTD 2012)*, IEEE, 2012, pp. 31-36. DOI: 10.1109/MTD.2012.6225997. See also: Letouzey, J.-L.; Ilkiewicz, M. "Managing Technical Debt with the SQALE Method". *IEEE Software* 29(6), 2012, pp. 44-51. https://ieeexplore.ieee.org/document/6225997

[4] Oman, P.; Hagemeister, J. "Construction and testing of polynomials predicting software maintainability". *Journal of Systems and Software* 24(3), 1994, pp. 251-266. DOI: 10.1016/0164-1212(94)90067-1. https://doi.org/10.1016/0164-1212(94)90067-1

[5] Heitlager, I.; Kuipers, T.; Visser, J. "A Practical Model for Measuring Maintainability". *QUATIC 2007*, IEEE, pp. 30-39. DOI: 10.1109/QUATIC.2007.7. Operationalization: *SIG/TÜViT Evaluation Criteria Trusted Product Maintainability*. https://dl.acm.org/doi/10.1109/QUATIC.2007.7 · https://www.softwareimprovementgroup.com/wp-content/uploads/SIG-TUViT-Evaluation-Criteria-Trusted-Product-Maintainability.pdf

[6] CMMI Institute / ISACA. *Capability Maturity Model Integration (CMMI) V3.0*, 2023. https://cmmiinstitute.com/ · https://cmmiinstitute.com/learning/appraisals/levels

[7] Forsgren, N.; Humble, J.; Kim, G. *Accelerate: The Science of Lean Software and DevOps*. IT Revolution Press, 2018. ISBN 978-1942788331. Research program: https://dora.dev/

[8] Inozemtseva, L.; Holmes, R. "Coverage Is Not Strongly Correlated with Test Suite Effectiveness". *ICSE 2014*, ACM. DOI: 10.1145/2568225.2568271. https://dl.acm.org/doi/10.1145/2568225.2568271

[9] Just, R.; Jalali, D.; Inozemtseva, L.; Ernst, M. D.; Holmes, R.; Fraser, G. "Are Mutants a Valid Substitute for Real Faults in Software Testing?". *FSE 2014*, ACM. DOI: 10.1145/2635868.2635929. https://dl.acm.org/doi/10.1145/2635868.2635929

[10] Rahman, F.; Devanbu, P. "How, and Why, Process Metrics Are Better". *ICSE 2013*, IEEE. https://ieeexplore.ieee.org/document/6606589/

[11] Nagappan, N.; Ball, T. "Use of Relative Code Churn Measures to Predict System Defect Density". *ICSE 2005*, ACM, pp. 284-292. DOI: 10.1145/1062455.1062514. https://dl.acm.org/doi/10.1145/1062455.1062514

[12] Bird, C.; Nagappan, N.; Murphy, B.; Gall, H.; Devanbu, P. "Don't Touch My Code! Examining the Effects of Ownership on Software Quality". *ESEC/FSE 2011*, ACM, pp. 4-14. DOI: 10.1145/2025113.2025119. https://dl.acm.org/doi/10.1145/2025113.2025119

[13] Avelino, G.; Passos, L.; Hora, A.; Valente, M. T. "A Novel Approach for Estimating Truck Factors". *ICPC 2016*, IEEE. https://arxiv.org/abs/1604.06766

[14] Krasner, H. / CISQ. *The Cost of Poor Software Quality in the US: A 2022 Report*. Consortium for Information & Software Quality, December 2022. https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2022-report/

[15] Pearce, H.; Ahmad, B.; Tan, B.; Dolan-Gavitt, B.; Karri, R. "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions". *IEEE Symposium on Security and Privacy (S&P) 2022*. https://arxiv.org/abs/2108.09293

[16] Perry, N.; Srivastava, M.; Kumar, D.; Boneh, D. "Do Users Write More Insecure Code with AI Assistants?". *ACM CCS 2023*. DOI: 10.1145/3576915.3623157. https://dl.acm.org/doi/abs/10.1145/3576915.3623157

[17] GitClear. *Coding on Copilot: 2023 Data Suggests Downward Pressure on Code Quality*. 2024. https://www.gitclear.com/coding_on_copilot_data_shows_ais_downward_pressure_on_code_quality

[18] GitClear. *AI Copilot Code Quality: 2025 Data Suggests 4x Growth in Code Clones*. 2025. https://www.gitclear.com/ai_assistant_code_quality_2025_research

[19] GitClear. *The Maintainability Gap: AI Code Quality in 2026*. January 2026. https://www.gitclear.com/the_ai_code_quality_maintainability_gap

[20] Peng, S.; Kalliamvakou, E.; Cihon, P.; Demirer, M. "The Impact of AI on Developer Productivity: Evidence from GitHub Copilot". arXiv:2302.06590, 2023 (preprint). https://arxiv.org/abs/2302.06590

[21] DORA / Google Cloud. *Accelerate State of DevOps Report 2024*. https://dora.dev/research/2024/dora-report/

[22] DORA / Google Cloud. *State of AI-assisted Software Development 2025*. September 2025. https://dora.dev/dora-report-2025/

[23] Fortune. "Google's code is now more than 25% AI-generated, CEO Sundar Pichai says". October 2024. https://fortune.com/2024/10/30/googles-code-ai-sundar-pichai/

[24] CNBC. "Satya Nadella says as much as 30% of Microsoft code is written by AI". April 2025. https://www.cnbc.com/2025/04/29/satya-nadella-says-as-much-as-30percent-of-microsoft-code-is-written-by-ai.html

[25] Semafor. "Google CEO says 75% of company's new code is AI-generated". April 2026. https://www.semafor.com/article/04/24/2026/google-ceo-says-75-of-companys-new-code-is-ai-generated

[26] Liu, Y.; Widyasari, R.; Zhao, Y.; Irsan, I. C.; Chen, J.; Lo, D. "Debt Behind the AI Boom: A Large-Scale Empirical Study of AI-Generated Code in the Wild". arXiv:2603.28592, March 2026 (preprint). https://arxiv.org/abs/2603.28592

[27] "Quality Assurance of LLM-generated Code: Addressing Non-Functional Quality Characteristics". arXiv:2511.10271, November 2025 (preprint). https://arxiv.org/abs/2511.10271

New in v2.0:

[28] OpenSSF. *SLSA — Supply-chain Levels for Software Artifacts*. Open Source Security Foundation. https://slsa.dev/

[29] Open Source Security Foundation. *OpenSSF Scorecard*. https://securityscorecards.dev/

[30] The White House. *Executive Order 14028: Improving the Nation's Cybersecurity*. May 2021 (established software bill of materials expectations for federal software supply). https://www.federalregister.gov/documents/2021/05/17/2021-10460/improving-the-nations-cybersecurity

[31] European Union. *Regulation (EU) 2024/2847 of 23 October 2024 — Cyber Resilience Act*, on horizontal cybersecurity requirements for products with digital elements. https://eur-lex.europa.eu/eli/reg/2024/2847/oj

[32] Google. *OSV — Open Source Vulnerabilities*. Distributed vulnerability database and schema. https://osv.dev/ · https://ossf.github.io/osv-schema/

[33] Joern — Open-source code analysis platform based on Code Property Graphs. Apache License 2.0. https://joern.io/ · https://github.com/joernio/joern

[34] GitHub. *CodeQL — Terms and Conditions*. Use restricted to open-source and academic/research contexts; automated analysis of others' code and hosted/commercial use are excluded without a separate agreement. https://github.com/github/codeql-cli-binaries · https://docs.github.com/en/code-security/codeql-cli

---