



A Systematic Review of AI-Assisted Code Generation in Regulated Software Environments: Reliability, Governance, and Compliance Implications

David Excel Ozowara

Goldman Sachs, Dallas, Texas
USA

Correspondence: daveozzy6@gmail.com

Keywords

AI-assisted code generation; large language models; software reliability; model risk management; regulatory compliance; software supply chain; automation bias; systematic literature review

Abstract

Large language models now write a material and growing share of production software, including software that executes regulated financial functions. The empirical literature on their reliability has grown rapidly and is, on its own terms, sophisticated. The governance frameworks that regulated institutions must satisfy were, without exception, written for a different kind of artefact, and in the most consequential recent case, were revised in full knowledge of generative AI and deliberately declined to cover it.

Objective. This review synthesises the evidence on the reliability of AI-assisted code generation, on the human factors governing its use, and on the intellectual-property and supply-chain exposures it creates; and determines how that evidence maps onto the model-risk, AI-specific and operational-resilience regimes applicable to regulated software environments.

Method. We conducted a systematic literature review following Kitchenham et al. and reported against PRISMA 2020, with a restricted multivocal stratum after Garousi et al. admitting regulatory instruments and supervisory guidance. Studies were appraised on an adapted Dybå–Dingsøyr instrument extended with an item recording whether deployment context or regulatory constraint is characterised.

Results. Five findings. First, benchmark performance and field reliability have decoupled: correctness benchmarks are saturated and contamination-prone while defect rates measured in real repositories remain substantial, so the headline capability numbers do not license deployment inferences. Second, security evidence is mixed, controlled model evaluations report high vulnerability rates while several developer studies find little or no security penalty, and the disagreement is explained by unit of analysis rather than by any study being wrong. Third, the most robust human-factors finding is not that assisted code is worse but that developers believe it is better than it is, which relocates the risk from the artefact to the review process. Fourth, package hallucination converts a model reliability property into an attacker-controllable supply-chain channel, and is the one exposure with no human-review mitigation. Fifth, and most consequentially, the April 2026 revision of US interagency model-risk guidance explicitly excludes generative and agentic AI from scope, leaving the primary supervisory framework silent on precisely this class of system.

Conclusion. We argue that AI-assisted code generation is misclassified when treated as a model-risk problem, because the model produces no decision and enters no inventory; it is better understood as a change to the software development lifecycle's assurance basis. We present the Generated Code Assurance Frame, a five-layer mapping from generation practice to the assurance claim it disturbs, and a research agenda centred on measuring review efficacy rather than generation quality.

Citation

Ozowara, D.E. (2026). A Systematic Review of AI-Assisted Code Generation in Regulated Software Environments: Reliability, Governance, and Compliance Implications. *Science, Education and Innovations in the Context of Modern Problems*, 9(10), 1–19. <https://doi.org/10.56334/sei/9.10.5>

Licensed

© 2026 The Author(s). Published by *Science, Education and Innovations in the Context of Modern Problems (SEI)*, under the auspices of IMCRA – International Meetings and Conferences Research Association (Azerbaijan).

This is an open access article distributed under the terms of the Creative Commons Attribution License (CC BY 4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

 <http://creativecommons.org/licenses/by/4.0/>

Received: June 11, 2026

Accepted: July 14, 2026

Published Online: September 20, 2026

1. INTRODUCTION

1.1 A capability that arrived before its governance

The adoption of AI coding assistants has been among the fastest of any development tool in the history of the field. Within roughly four years of the first widely available commercial assistant, generated code has moved from novelty to default in many organisations, including organisations whose software executes regulated financial functions; credit decisioning, payment processing, transaction reporting, capital calculation.

The research response has been correspondingly rapid and is, on its own terms, strong. There is now a mature benchmark literature measuring functional correctness [8,9,10,11,12], a body of work measuring security properties of generated code under controlled prompting [23,29], field studies measuring defects in real repositories [25,30], controlled developer studies measuring the effect of assistance on outcomes [24,26,36,38], and an emerging supply-chain literature on a failure mode with no analogue in human-written code [33,34].

The governance response has not kept pace, and in one specific and important respect it has moved in the opposite direction. In April 2026 the US federal banking agencies issued revised interagency guidance on model risk management, superseding the 2011 guidance that had defined supervisory expectations for fifteen years [52,53]. The revision was made with full awareness of generative AI, and it explicitly places generative and agentic AI outside its scope on the grounds that these technologies are novel and rapidly evolving [53]. The effect is that the primary US supervisory framework for managing risk from computational systems in banking now says, in terms, that it does not cover this class of system. The EU AI Act imposes obligations that bite hard on AI systems making regulated *decisions* [51], and the European Banking Authority has begun mapping its interaction with existing sectoral requirements [59], but an assistant that helps an engineer write the credit-decisioning system is not itself the credit-decisioning system, and falls outside Annex III on a straightforward reading.

So there is a gap, and it is not an accident of timing. It is a gap that opened because AI-assisted code generation does not fit the shape of any existing regime: it is not a model producing a decision, it is not a third-party service processing customer data, and it is not a change to the production estate. It is a change to *how software is made*, and no financial-services regulatory instrument governs that directly.

1.2 Why the research literature does not close the gap either

One might expect the software-engineering literature to fill the space. It does not, for a reason that is structural rather than a matter of maturity.

Almost the entire reliability literature measures the *artefact*: is the generated code correct, is it secure, does it pass tests. This is the right measurement for a research programme aimed at improving models. It is the wrong measurement for a governance question, because in every regulated deployment the generated code is not shipped directly. It passes through review, testing, static analysis and change control. The question that determines regulatory exposure is therefore not "how good is generated code" but "how much of the defect rate survives the assurance process, and has the assurance process's effectiveness changed now that its inputs are machine-produced at a higher rate."

That second question is barely studied. Our extraction found abundant measurement of generation quality, meaningful measurement of developer outcomes, and almost no measurement of review efficacy on generated code specifically.

1.3 Research questions

- **RQ1.** What does the evidence show about the functional reliability of AI-generated code, and how well do benchmark measures predict field behaviour?
- **RQ2.** What does the evidence show about the security properties of AI-generated code, and how are apparently conflicting findings reconciled?
- **RQ3.** What do human-factors studies show about developer behaviour, trust and review when using assistants?
- **RQ4.** What intellectual-property and supply-chain exposures are documented?
- **RQ5.** How do these map onto applicable governance regimes, and what research agenda follows?

1.4 Contributions

1. A reconciliation of the apparently contradictory security evidence, showing that the disagreement is a unit-of-analysis artefact rather than a genuine empirical conflict.
2. Identification of **review efficacy** as the governing variable and as the field's principal measurement gap.
3. An account of the **regulatory perimeter problem**: AI code assistance falls outside model-risk guidance by explicit exclusion, outside the AI Act's high-risk list by construction, and outside third-party regimes by category.
4. The **Generated Code Assurance Frame (GCAF)**, mapping generation practice to the specific assurance claim it disturbs.

2. BACKGROUND AND RELATED WORK

2.1 From code completion to code generation

Transformer models trained on public code repositories made functional program synthesis from natural-language description practical, a capability first characterised systematically alongside the HumanEval benchmark [8] and rapidly extended by MBPP [9] and APPS [10]. The subsequent trajectory is documented comprehensively in Jiang

et al.'s survey, which covers data curation, evaluation, ethical and environmental dimensions [19], and the open-weight lineage is represented by Code Llama [20] and StarCoder2 [21].

For this review the important structural fact is that these models are trained predominantly on public open-source code, which determines both their competence profile and several of their failure modes: they are strongest where public examples are dense, and they reproduce the security and quality characteristics of the corpus they learned from.

2.2 The assurance context they enter

Regulated software development is not an unassured process into which generated code arrives. It has an established assurance apparatus: peer code review, whose effectiveness and limits are well documented [63,64,65]; automated testing; static analysis; change control; and, for financial institutions, segregation of duties and auditable approval. The defect-reduction literature establishes the baseline that review and testing catch a substantial but far from complete proportion of defects, with effectiveness varying by defect class [66].

Two adjacent literatures matter for interpretation. Work on engineering machine-learning systems documents how ML components erode conventional software assurance assumptions [69] and accumulate distinctive forms of technical debt [70], and the safety-engineering tradition supplies the framing that automation typically does not remove human error but relocates it [45,67,68].

2.3 Prior secondary studies

Jiang et al.'s TOSEM survey is the most complete secondary treatment of LLM code generation [19]. A recent PRISMA-guided review synthesises technical and demographic dimensions across 58 studies [22]. On the supply-chain side, systematic treatment of package hallucination has begun to appear. We identified no secondary study that takes the *regulated deployment* as its unit of analysis, and none that treats the interaction between generation practice and the existing software assurance process as its central object.

3. METHOD

3.1 Protocol and reporting

We followed the systematic review guidelines of Kitchenham et al. [1] with the mapping conventions of Petersen et al. [2], reported against the PRISMA 2020 statement [7], and applied a restricted multivocal extension after Garousi et al. [3] to admit regulatory instruments and supervisory guidance. Snowballing followed Wohlin [4].

A methodological caution specific to this domain deserves statement up front. This is a fast-moving literature in which the studied artefacts (the models) change materially between a study's execution and its publication. A finding about a 2022 model is not straightforwardly a finding about the technology; it is a finding about a specific system at a specific version. We therefore extracted *model identity and version* as a first-class field and treat cross-study comparison with corresponding caution. Sallou et al.'s analysis of threats to validity in LLM-based software engineering research informed this decision [22].

3.2 Search strategy

Sources: ACM Digital Library, IEEE Xplore, ScienceDirect, SpringerLink, Scopus and USENIX proceedings, with arXiv admitted for this review beyond preprint location, because a substantial fraction of the primary evidence in this fast-moving area appears there first; arXiv-only records were flagged as such in extraction and are identified in the reference list. Concept blocks:

- **Block A (generation):** ("code generation" OR "code completion" OR "code assistant" OR Copilot OR "program synthesis" OR "coding agent")
- **Block B (model):** ("large language model" OR LLM OR "generative AI" OR transformer OR Codex)
- **Block C (property):** (reliability OR correctness OR security OR vulnerabilit* OR quality OR trust OR governance OR compliance OR licens* OR "supply chain")

Coverage from 2021, the year of the first widely studied model, to the search execution date.

3.3 Inclusion and exclusion

Inclusion. (I1) peer-reviewed study, or arXiv preprint meeting I3, or admissible instrument per §3.4; (I2) addresses generation, evaluation, use or governance of machine-generated source code; (I3) reports empirical method, evaluation or systematic argument; (I4) English; (I5) full text retrievable.

Exclusion. (E1) editorials, abstracts, posters; (E2) vendor material outside §3.4; (E3) work on code *understanding* or *summarisation* with no generation component; (E4) duplicates, retaining the most complete version; (E5) model or tool announcements without evaluation; (E6) leaderboard reports without methodological description.

3.4 Regulatory and standards stratum

Admissible: legislative and regulatory instruments in force; supervisory guidance issued by a named financial authority; standards issued by a recognised standards body; and guidance from a national technical authority. Vendor and consultancy material was excluded from the evidential base. This exclusion is applied strictly here because the commercial grey literature on AI coding tools has an exceptionally strong directional bias, being produced predominantly by parties selling the tools or selling remediation for them.

3.5 Quality appraisal

Studies were appraised on the eleven-item instrument used in this review series, adapted from Dybå and Dingsøyr [5], with the eleventh item recording whether the study characterises *deployment context*, whether generated code was reviewed, tested, or shipped, and under what constraints. As in the companion reviews, item eleven is an outcome measure rather than an exclusion criterion, and its distribution supports the central claim of §4.4 and §7.

3.6 Extraction and synthesis

Extraction covered: model identity and version; task type; evaluation instrument; unit of analysis (model output, developer, repository, organisation); reported effect; deployment context; and regulatory treatment. Synthesis was thematic. Quantitative pooling was not attempted: reported rates are not commensurable across studies because they differ in model, prompt construction, language, defect taxonomy, detection tooling and, decisively, in unit of analysis, which §5.3 shows to be the source of the field's apparent contradictions.

3.7 Reproducibility

The strings, criteria, appraisal instrument and extraction schema permit re-execution. Screening counts are not reported: in this domain especially, database contents change weekly, and counts a reader cannot verify would convey false precision. Replicators should record and report their own flow.

4. RQ1: FUNCTIONAL RELIABILITY

4.1 The benchmark lineage and what it measures

Execution-based benchmarking is the field's dominant evaluation paradigm. HumanEval established the pattern, hand-written function-level problems with hidden unit tests, scored by pass@k [8], and MBPP [9] and APPS [10] extended it in scale and difficulty. MultiPL-E generalised across languages [11]. The paradigm's virtue is that it measures something real and objectively checkable; its limitations were exposed from within the field.

EvalPlus demonstrated that HumanEval's original test suites were insufficient, and that augmenting them by roughly two orders of magnitude in test count causes a substantial fraction of previously "passing" solutions to fail [12]. This is the single most important methodological result in the correctness literature, because it establishes that reported pass rates are a function of test adequacy rather than a property of the code, and that the field's headline numbers were systematically optimistic.

The second structural problem is contamination. Benchmarks published before a model's training cut-off may be present in its training data, converting an evaluation of synthesis into a partial evaluation of recall. LiveCodeBench addresses this by continuously admitting problems postdating model releases, and reports that models performing comparably on HumanEval separate materially on contamination-controlled problems, with fine-tuned open models disproportionately represented among those that do worse [13]. EvoEval attacks the same issue by systematically transforming existing problems into novel ones [15].

The third is scope. Function-level synthesis is not software engineering. SWE-bench evaluates resolution of real repository issues [14], BigCodeBench targets library-intensive tasks requiring diverse API composition [16], ClassEval moves to class level [17] and DevEval to repository level [18]. The consistent pattern across this progression is that measured capability falls sharply as the task approaches realistic scope.

4.2 The saturation problem

Frontier models now score in the low-to-mid nineties on HumanEval. Read naively this suggests near-solved function synthesis. Read against §4.1 it suggests the benchmark has ceased to discriminate: with insufficient tests [12], probable contamination [13] and unrepresentative scope [14,16,18], a high score is consistent with a wide range of true reliability.

The governance consequence is direct and important. **Benchmark scores cannot support a deployment argument in a regulated environment.** An institution that justified reliance on generated code by citing benchmark performance would be resting an assurance claim on a measure its own producers have shown to be saturated, contaminated and out of scope. We are not aware of any published work establishing a defensible mapping from any correctness benchmark to a field defect rate.

4.3 Non-functional properties

Correctness is not the binding constraint in most regulated code. Maintainability, efficiency, error handling and adherence to institutional standards all matter, and the emerging non-functional literature reports that functionally correct generated code frequently falls short on them [32]. Static-analysis-driven iterative prompting improves several such dimensions measurably [32], which is a practically useful result and also an implicit admission: the improvement is necessary because the default output does not meet the standard.

4.4 The deployment-context gap

Applying appraisal item eleven across the reliability stratum produces a stark distribution. The overwhelming majority of correctness studies evaluate model output in isolation: generated, scored, discarded. They do not report whether the code was reviewed, whether review caught the defects, or whether anything was shipped.

This is entirely appropriate for the purpose these studies serve, which is comparing models. It means, however, that the reliability literature (however large) cannot answer the question a regulated institution actually faces, which concerns residual defect rate after assurance, not raw generation quality.

Finding 1. *Benchmark performance and field reliability have decoupled. Correctness benchmarks are saturated, contamination-prone and scoped below realistic engineering tasks, and no published mapping exists from benchmark score to field defect rate. Benchmark performance cannot support a deployment assurance argument.*

5. RQ2: Security Properties

5.1 The controlled-generation evidence

The foundational study is Pearce et al.'s systematic prompting of a commercial assistant across scenarios derived from the MITRE CWE Top 25, producing 1,689 programs of which approximately 40% contained vulnerabilities, with rates varying by language and weakness class [23]. Targeted replication on later model versions found measurable improvement in some categories while confirming that vulnerable suggestions remained common across a diversity of weaknesses and scenarios [31]. Khoury et al. found comparable results prompting a general-purpose chat model, with secure output in a minority of security-relevant tasks [28], and Siddiq and Santos contributed a purpose-built evaluation dataset for the problem [29].

5.2 The field evidence

Field studies analyse code actually committed to repositories. Fu et al.'s study identified 733 snippets attributable to three assistants in GitHub projects and found security weaknesses in roughly 29.5% of Python and 24.2% of JavaScript snippets, spanning many CWE categories including several from the Top 25 [25]. A larger-scale analysis of 7,703 files explicitly attributed to four tools, scanned with static analysis, reported 4,241 CWE instances across 77 vulnerability types while finding that 87.9% of files contained no identifiable CWE-mapped vulnerability [30].

These two field numbers look inconsistent and are not. They differ in unit, snippet versus file, in attribution method, in language mix and in detection tooling. A per-snippet rate near a quarter and a per-file clean rate near ninety percent are compatible if most files contain few generated snippets. The lesson is that a single "vulnerability rate for AI code" does not exist as a well-defined quantity.

5.3 The developer-study evidence and the apparent contradiction

The most-cited human-subjects result is Perry et al.'s controlled study of 47 participants across five security-relevant tasks in three languages, finding that participants with assistant access wrote significantly less secure code, with particularly strong effects on encryption and injection tasks [24]. Yet Sandoval et al.'s user study found that assisted participants produced critical security bugs at a rate not materially above control, suggesting no significant increase in severe defects [26], and Asare et al., comparing assistant output against the human-introduced vulnerabilities in a historical dataset, found the assistant not uniformly worse than the humans, and on harder tasks associated with *more* secure solutions [27].

The literature frequently presents these as conflicting. We argue they are not, and that the reconciliation is a unit-of-analysis distinction that the field has not made explicit:

- Studies asking "**is generated code less secure than a correct implementation?**" consistently answer yes [23,28,31].
- Studies asking "**is generated code less secure than what a comparable human would have written?**" answer *not clearly*, and sometimes no [26,27].

- Studies asking "**do developers with assistants produce less secure output than developers without?**" answer yes under some task and model conditions [24] and not under others [26].

These are three different questions with three different baselines, and all three sets of findings can be simultaneously true. The confusion arises from a shifting implicit comparator.

For a regulated institution, the second question is the operative one, and its answer, *roughly comparable to human baseline*, is far less reassuring than it sounds, because the human baseline is not a standard the institution has ever accepted. The assurance apparatus of §2.2 exists because unreviewed human code is not acceptable. Parity with an unacceptable baseline, delivered at substantially higher volume, is a throughput problem for the review process rather than a reason for comfort.

Finding 2. *The apparent contradiction in the security literature is a unit-of-analysis artefact, resolved by making the comparator explicit. Generated code is less secure than correct code, roughly comparable to human-written code, and produced at higher volume, which converts a code-quality question into a review-capacity question.*

5.4 The volume argument, stated precisely

If generated code has a defect rate comparable to human-written code, and assistance increases code production rate, then absolute defect volume entering review increases proportionally. Review capacity, meanwhile, is bounded by human attention and does not scale with generation. Unless review throughput increases commensurately or per-unit review effectiveness improves, residual defect rate in shipped code rises even though per-unit code quality is unchanged.

This is an arithmetic consequence of the empirical findings, not a separate empirical claim, and we found no study that tests it directly. It is the single most policy-relevant untested proposition in the corpus.

6. RQ3: Human Factors, Trust and Review

6.1 Productivity effects

The productivity literature reports positive but heterogeneous effects. Peng et al.'s controlled experiment found assisted participants completed an HTTP-server task substantially faster than controls [36]. Ziegler et al.'s large-scale telemetry study across many developers found perceived productivity correlated with suggestion acceptance rate rather than with coding iteration time, which is a more subtle result than it first appears: it means the felt benefit and the measured benefit are not the same quantity [37]. Vaithilingam et al. found that participants often failed tasks more frequently with an assistant than with conventional completion, owing to incorrect suggestions, while nonetheless *preferring* the assistant [38]. Imai reported higher productivity but lower code quality relative to human pair programming [39].

The consistent shape across these studies is a divergence between measured outcome and perceived benefit, with perception running ahead. This is the thread that connects the productivity literature to the security literature.

6.2 Overconfidence and misplaced trust

Perry et al.'s second finding is more important for governance than their first: participants with assistant access were *more likely to believe their code was secure*, including when it was not [24]. Participants who trusted the assistant less and engaged more actively with prompt construction produced fewer vulnerabilities, that is, the moderating variable was scepticism.

Barke et al.'s qualitative study identifies two interaction modes, acceleration, where the developer knows what to write and the assistant speeds it up, and exploration, where the developer does not and the assistant supplies direction

[40]. The distinction matters because verification burden differs sharply between them: in acceleration the developer has an internal correctness model to check against, and in exploration they do not. Prather et al.'s study of novice programmers documents a widening gap in which the tools help those who can already evaluate output and harm those who cannot [41], and Sabouri et al. characterise how trust in assistants forms and shifts over time [42].

6.3 The automation-bias frame

This pattern is not novel to AI coding; it is the well-documented behaviour of humans supervising automation. Parasuraman and Riley's account of use, misuse, disuse and abuse of automation [43] and Skitka et al.'s experimental work on automation bias [44] establish that operators systematically under-detect automation errors, particularly errors of omission, and that the effect strengthens as automation reliability increases, because reliable automation trains the operator to stop checking. Bainbridge's ironies of automation makes the structural point: automating the routine parts of a task leaves humans responsible for the same residual cases they now have least practice at handling [45].

Applied here, the implication is uncomfortable and directly contrary to intuition. **Improvements in model reliability may reduce review effectiveness faster than they reduce defect rate.** A model that is wrong 30% of the time keeps reviewers alert; a model that is wrong 3% of the time trains them to approve. Nothing in the reviewed corpus tests this, and it is the mechanism most likely to produce a serious regulated-sector incident.

6.4 The review-efficacy gap

Modern code review's effectiveness and limits are well characterised for human-authored code: review coverage and participation correlate with post-release quality [63], reviewers pursue multiple goals of which defect-finding is only one [64], and at scale review functions substantially as knowledge transfer and lightweight consistency checking rather than exhaustive verification [65].

None of this has been re-measured for generated code, and there are strong reasons to expect it to differ. Generated code is often longer and more fluent than a human would write, which is known to reduce review scrutiny. It carries no author to interrogate about intent. It may be reviewed by the developer who prompted it, who has already implicitly endorsed it. And it arrives faster, compressing available review time per unit.

Finding 3. *The best-supported human-factors result is not that assisted code is worse but that developers are more confident in it than warranted. This relocates the risk from the artefact to the review process (which is the part of the pipeline the literature has not measured).*

7. RQ4: INTELLECTUAL PROPERTY AND SUPPLY-CHAIN EXPOSURE

7.1 Memorisation and provenance

Language models memorise training data, and memorised sequences can be extracted [46], with memorisation increasing in model scale, data duplication and context length [47]. For code models trained on public repositories, this creates a specific exposure: generated code may reproduce training material verbatim or near-verbatim, carrying licence obligations that the receiving organisation neither knows about nor has satisfied. Ciniselli et al. measured the extent to which code recommenders produce predictions by cloning from the training set [48], and Al-Kaswan et al. documented traces of memorisation in code models specifically [49], following earlier work framing the licensing question directly [50].

For a regulated institution the exposure is not primarily litigation risk. It is that **provenance is unknown**. Software composition analysis, SBOM generation and licence compliance all presuppose that the origin of code is

determinable. Generated code has no recorded origin: it is neither written in-house in the ordinary sense nor imported as an identified dependency, and it enters the codebase invisible to the tooling built to track exactly this. The in-toto framework for supply-chain integrity [35] assumes a chain of attested steps; generation inserts a step that attests nothing.

7.2 Package hallucination

The most concrete and best-quantified supply-chain exposure is package hallucination: models recommend dependencies that do not exist. Spracklen et al.'s study across 16 code-generating models, generating hundreds of thousands of Python and JavaScript samples, found that roughly a fifth of recommended packages were non-existent, with open-weight models averaging materially higher rates than commercial ones, and identified over two hundred thousand unique fabricated package names [33].

The property that converts this from a nuisance into an attack is *persistence*: a substantial proportion of hallucinated names recur across repeated prompting rather than being sampled randomly. That makes the fabricated names predictable, and predictability makes them registrable. An adversary who enumerates a model's recurring hallucinations can pre-register those names in a public registry with malicious content: the attack pattern now termed slopsquatting. Zhao et al.'s fuzzing approach demonstrates that hallucinated package names can be systematically elicited [34], which is simultaneously a testing contribution and a demonstration of attacker capability.

Package hallucination has a property that distinguishes it from every other exposure in this review, and that deserves emphasis. **It is the one failure mode that human code review does not catch.** A reviewer reading an import statement for a plausibly named library has no signal that the library is fabricated; the code looks correct, and if an attacker has registered the name, the dependency resolves and installs successfully. Detection requires registry-level verification against an allow-list, which is a build-pipeline control rather than a review control. The mitigation is therefore mandatory rather than advisory in any regulated pipeline.

Finding 4. *Package hallucination converts a model reliability property into an attacker-controllable supply-chain channel, is quantified at roughly one in five recommended packages across studied models, persists predictably across prompts, and is the sole documented exposure with no human-review mitigation.*

8. RQ5: THE GOVERNANCE MAPPING AND THE PERIMETER PROBLEM

8.1 Model risk management, and the explicit exclusion

The dominant framework for governing computational systems in banking is model risk management. The 2011 US interagency guidance defined a model as a quantitative method producing estimates, and organised expectations around development, validation and governance, requiring model inventory, independent validation, ongoing monitoring and effective challenge [52]. The UK Prudential Regulation Authority's supervisory statement establishes closely comparable expectations [54].

Applying this to an AI coding assistant produces immediate category failure. The assistant produces no estimate and makes no decision; its output is source code, which is then modified, reviewed and deployed by humans. It does not belong in a model inventory in any conventional reading, and independent validation of it is not a well-formed activity; one does not validate a text editor.

The April 2026 revision resolves this ambiguity in the least helpful direction available. The revised guidance preserves the core principles of validation, monitoring, governance and effective challenge while adopting a more materiality-sensitive posture, and it states that generative and agentic AI models are novel and rapidly evolving and are therefore

not within its scope [53]. This is a deliberate choice, made with full knowledge of the technology, and its consequence is that the primary US supervisory framework for computational risk in banking is now explicitly silent on generative AI, including, a fortiori, on generative AI used to construct the systems that the framework *does* cover.

8.2 AI-specific regimes, and why the perimeter misses

The EU AI Act classifies systems by intended purpose, and Annex III places creditworthiness assessment and comparable financial applications in the high-risk category, attracting conformity assessment, technical documentation, data governance, logged human oversight and post-market monitoring [51,59]. The obligations are substantial and, for a bank deploying AI in customer decisions, decisive.

They do not reach a coding assistant. The assistant's intended purpose is software development, which is not an Annex III use. On a straightforward reading, an assistant used to write a high-risk credit-scoring system is not itself high-risk, notwithstanding that a defect it introduces would manifest inside a high-risk system. Voluntary frameworks, the NIST AI Risk Management Framework [55] and ISO/IEC 42001 [56], are broad enough in principle to cover development-time AI, but they are voluntary, and neither supplies the specific controls this application requires.

8.3 Where the obligation actually lands

The obligation does not disappear; it lands elsewhere, and less visibly.

Under DORA, ICT risk management covers the systems supporting critical or important functions, and an assistant integrated into the development pipeline for those systems is part of the ICT estate and, where externally provided, a third-party dependency with contractual and oversight consequences [57]. The operational-resilience principles require institutions to understand and control the dependencies supporting critical operations [61], which reaches development-time dependencies on a purposive reading. Secure-development guidance such as the NIST Secure Software Development Framework specifies practices, provenance verification, dependency management, review, that apply regardless of who or what authored the code [58]. And where generated code participates in producing a reported risk figure, the data-lineage expectations of BCBS 239 apply to the resulting derivation [60].

The pattern is that AI code assistance is governed *indirectly*, through obligations attaching to the systems it helps build, rather than *directly*, through any framework addressing the practice. The supervisory record indicates how this ends: the TSB enforcement turned on governance, testing adequacy and supplier assurance rather than on any technical deficiency [62], and a defect introduced by generation, surviving review and reaching production would be characterised in exactly those terms.

Finding 5. *AI-assisted code generation sits outside every applicable direct regime, excluded from model-risk guidance explicitly, from the AI Act's high-risk list by construction, and from third-party regimes by category, while remaining fully within the indirect obligations attaching to the systems it builds. Institutions are accountable for an activity that no framework tells them how to control.*

9. SYNTHESIS: THE GENERATED CODE ASSURANCE FRAME

9.1 The reclassification

The organising argument of this paper is that AI-assisted code generation is *misclassified* when treated as a model-risk problem. It is not a model producing a decision. It is a change to the assurance basis of the software development lifecycle: the process by which an institution convinces itself, and its supervisors, that its software does what it is supposed to.

Every assurance claim an institution makes about its software rests on premises. Those premises were formed when code was written by identified humans at human speed. Generation does not falsify them all, but it disturbs several, and the disturbed ones are not obvious without being enumerated. GCAF enumerates them.

9.2 The five layers

L1. Authorship and intent. *Prior premise:* every line has an author who can be asked why it is there, and whose judgement is part of the assurance chain. *What generation disturbs:* the prompter is not the author in this sense; they selected an output whose rationale is not recoverable. *Evidence required:* recorded association between generated segments, the prompt and model version that produced them, and the identified engineer who accepted them. *Literature support:* none; provenance recording for generated code is unaddressed in the corpus.

L2. Reliability basis. *Prior premise:* competence is evidenced by qualification, experience and track record. *What generation disturbs:* the substitute offered is benchmark performance, which §4.2 shows cannot bear the weight. *Evidence required:* defect rates measured on the institution's own code in its own domain, not benchmark scores. *Literature support:* extensive but non-transferable [8–19].

L3. Review efficacy. *Prior premise:* review catches a known and adequate proportion of defects. *What generation disturbs:* input volume rises (§5.4) while reviewer scrutiny plausibly falls (§6.3), and neither effect has been measured. *Evidence required:* measured review catch-rate on generated versus authored code; review time per unit; defect escape rate by origin. *Literature support:* strong for human code [63,64,65], absent for generated code, **the field's central gap**.

L4. Provenance and dependency integrity. *Prior premise:* every component's origin is determinable and its licence known; every dependency is a real, vetted artefact. *What generation disturbs:* generated code has no recorded origin [46–50], and hallucinated dependencies are attacker-registrable [33,34]. *Evidence required:* dependency allow-listing with registry verification at build time; SBOM extended to record generation events; licence scanning tuned for near-verbatim reproduction. *Literature support:* the exposure is well quantified [33]; the controls are not evaluated.

L5. Accountability. *Prior premise:* a named individual is responsible for each change, which is what makes segregation of duties and management attestation meaningful. *What generation disturbs:* responsibility diffuses across prompter, reviewer, model provider and training corpus, and the diffusion is invisible in change records. *Evidence required:* explicit policy assigning accountability to the accepting engineer, with generation recorded but not exculpatory; management-body awareness of generation prevalence in critical systems. *Literature support:* none in the technical corpus; the governance literature has not reached the question.

9.3 Reading the frame

The layers are ordered by increasing distance from where the research effort sits. L2 is the most studied layer in the entire field and is the one whose evidence transfers least. L3 is the layer that actually determines residual risk and is essentially unstudied. L5 is where an enforcement action would land and is unaddressed by anyone.

GCAF is a synthesis instrument derived from the reviewed evidence, not an evaluated artefact. No reviewed study implements or measures anything of this shape, and its layers should be read as hypotheses about where assurance breaks rather than as validated findings.

10. DISCUSSION AND RESEARCH AGENDA

10.1 What follows from the reclassification

If AI code generation is a change to the assurance basis rather than a new model to govern, then three things follow.

The right unit of governance is the **pipeline**, not the tool. Controls attach to how generated code enters, is verified and is recorded; not to approving or forbidding an assistant.

The right measurement is **residual defect rate after assurance**, not generation quality. An institution that knows only its model's benchmark score knows nothing decision-relevant.

The right locus of concern is **review**, because review is the only layer that converts generation quality into shipped quality, and it is the layer under the most pressure and the least measurement.

10.2 Research agenda

A1. Measure review efficacy on generated code. The field's central gap. A controlled study comparing reviewer defect-detection rates, review duration and confidence on matched generated and human-authored code would be tractable, high-impact, and is not, to our knowledge, published. Extending the code-review measurement tradition [63,64,65] is the obvious method.

A2. Test the reliability–vigilance inversion. §6.3 predicts that increasing model reliability decreases reviewer scrutiny, potentially faster than it decreases defect rate. This is directly testable by varying injected defect rates across review conditions, and the automation-bias literature supplies the design [43,44].

A3. Establish benchmark-to-field transfer, or establish that it fails. Determine empirically whether any correctness benchmark predicts field defect rate in production codebases. A negative result is as valuable as a positive one, since it would settle whether benchmark scores may be cited in assurance arguments at all.

A4. Evaluate provenance recording for generated code. Design and evaluate mechanisms binding generated segments to prompt, model version and accepting engineer, with acceptable developer friction. This is the enabling control for L1, L4 and L5 simultaneously, and would extend supply-chain attestation frameworks [35] to cover a step they currently do not model.

A5. Evaluate hallucination controls at pipeline scale. Registry verification, dependency allow-listing and pre-resolution checks are the mitigations implied by §7.2; none is evaluated for coverage, false-positive rate or developer cost in the reviewed corpus.

A6. Longitudinal study of a regulated estate. Track generation prevalence, defect escape rate and review metrics across a regulated codebase over time. The gap this addresses is the same one identified in the companion reviews: no published study observes any of these phenomena inside a supervised institution.

A7. Regulatory-perimeter analysis. Legal and policy scholarship on whether development-time AI should be brought within model-risk or AI-specific regimes, and what the marginal supervisory benefit would be. The April 2026 exclusion [53] was a deliberate choice and deserves analysis rather than assumption.

10.3 Implications for practice

Six propositions follow from the evidence, each traceable above. Govern the pipeline, not the tool. Do not cite benchmark performance in an assurance argument (§4.2). Treat review capacity as the binding constraint and resource it in proportion to generation volume (§5.4). Make registry verification mandatory rather than advisory, because it is the only control for the one exposure review cannot catch (§7.2). Record provenance now, because retrofitting it across a codebase is far more expensive than capturing it at generation time. And assign accountability explicitly to the accepting engineer, since diffusion of responsibility is the condition under which the automation-bias effects of §6.3 become organisationally invisible.

11. THREATS TO VALIDITY

We classify threats following the established taxonomy for validity in software-engineering secondary studies [6], with one addition specific to this domain.

Construct validity. "AI-generated code" spans single-line completion, whole-function synthesis and autonomous agent output: practices with materially different assurance implications that the literature frequently conflates, and that we could only partly separate through extraction. "Vulnerability rate" is not a well-defined quantity across studies (§5.2). The constructs *review efficacy* and *assurance basis*, and the GCAF built on them, are introduced here and unvalidated by prior use.

Internal validity. The review was designed around a hypothesised governance gap, risking confirmation bias; we mitigated by pre-specifying appraisal item eleven so the deployment-context claim rests on systematic coding. The reconciliation in §5.3 is an interpretive contribution, and readers may reasonably weigh the underlying studies differently.

Temporal validity, the dominant threat here. Model capabilities change faster than the publication cycle. Several central results were obtained on models now multiple generations superseded, and it is unclear whether they describe the technology or specific historical systems. We extracted model version to make this visible, but the threat cannot be eliminated, and readers should treat every quantitative rate in this review as version-specific. This is a stronger caveat than we would attach to either companion review.

External validity. The regulatory analysis is weighted toward EU, UK and US financial supervision. The technical findings should transfer to other regulated sectors, though the applicable instruments differ. Most primary studies use open-source or synthetic tasks; proprietary financial codebases may differ systematically in ways no reviewed study measures.

Conclusion validity. No pooling was attempted and none would be defensible (§3.6). The claim that gaps exist is better supported than any claim about what filling them would show. The volume argument of §5.4 is arithmetic given the empirical premises, not an independently tested finding, and is labelled as such.

Publication and source bias. Substantial primary evidence is arXiv-only and has not passed peer review; we admitted it because excluding it would have produced a corpus systematically lagging the technology, but flagged it in extraction. Negative results, organisations that adopted assistants and reverted, are essentially unpublishable, and their absence should be assumed rather than inferred against.

12. CONCLUSION

The literature on AI-assisted code generation is large, fast-moving and, on its own terms, rigorous. This review finds that it measures the wrong thing for the question regulated institutions face: not through any failing of the researchers, but because a literature aimed at improving models naturally measures model output, while an institution's exposure is determined by what survives its assurance process.

The evidence supports a coherent picture once the comparators are made explicit. Generated code is less secure than correct code and roughly comparable to human-written code, produced at higher volume, reviewed by developers who are more confident in it than they should be, carrying no recorded provenance, and occasionally importing dependencies that an attacker has been waiting to register. Each element is individually manageable. Together they constitute a shift in how software assurance works, and the shift is happening inside institutions that are accountable

for their software and have no framework telling them how to control it, a gap that the most recent revision of the relevant supervisory guidance made wider rather than narrower, by design.

The most valuable next study is also the most tractable. Measure whether code review catches defects in generated code at the rate it catches them in human-written code. Everything else in the assurance chain is downstream of that number, and at present nobody knows what it is.

REFERENCES

1. Kitchenham, B., Pearl Brereton, O., Budgen, D., Turner, M., Bailey, J., & Linkman, S. (2009). Systematic literature reviews in software engineering – A systematic literature review. *Information and Software Technology*, 51(1), 7–15.
2. Petersen, K., Vakkalanka, S., & Kuzniarz, L. (2015). Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64, 1–18.
3. Garousi, V., Felderer, M., & Mäntylä, M. V. (2019). Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology*, 106, 101–121.
4. Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. ACM.
5. [5] Dybå, T., & Dingsøyr, T. (2008). Empirical studies of agile software development: A systematic review. *Information and Software Technology*, 50(9–10), 833–859.
6. Zhou, X., Jin, Y., Zhang, H., Li, S., & Huang, X. (2016). A map of threats to validity of systematic literature reviews in software engineering. In *Proceedings of the 23rd Asia-Pacific Software Engineering Conference (APSEC)*. IEEE.
7. Page, M. J., McKenzie, J. E., Bossuyt, P. M., Boutron, I., Hoffmann, T. C., Mulrow, C. D., et al. (2021). The PRISMA 2020 statement: An updated guideline for reporting systematic reviews. *BMJ*, 372, n71.
8. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. de O., Kaplan, J., et al. (2021). *Evaluating large language models trained on code*. arXiv:2107.03374.
9. Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., et al. (2021). *Program synthesis with large language models*. arXiv:2108.07732.
10. Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., et al. (2021). Measuring coding challenge competence with APPS. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*.
11. Cassano, F., Gouwar, J., Nguyen, D., Nguyen, S., Phipps-Costin, L., Pinckney, D., et al. (2023). MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7), 3675–3691.
12. Liu, J., Xia, C. S., Wang, Y., & Zhang, L. (2023). Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems (NeurIPS)*.

13. Jain, N., Han, K., Gu, A., Li, W.-D., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., & Stoica, I. (2025). LiveCodeBench: Holistic and contamination-free evaluation of large language models for code. In *International Conference on Learning Representations (ICLR)*.
14. Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. (2024). SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*.
15. Xia, C. S., Deng, Y., & Zhang, L. (2024). *Top leaderboard ranking = top coding proficiency, always? EvoEval: Evolving coding benchmarks via LLM*. arXiv:2403.19114.
16. Zhuo, T. Y., Vu, M. C., Chim, J., Hu, H., Yu, W., Widiasari, R., et al. (2025). BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. In *International Conference on Learning Representations (ICLR)*.
17. Du, X., Liu, M., Wang, K., Wang, H., Liu, J., Chen, Y., et al. (2023). *ClassEval: A manually-crafted benchmark for evaluating LLMs on class-level code generation*. arXiv:2308.01861.
18. Li, J., Li, G., Zhao, Y., Li, Y., Liu, H., Zhu, H., et al. (2024). *DevEval: A manually-annotated code generation benchmark aligned with real-world code repositories*. arXiv:2405.19856.
19. Jiang, J., Wang, F., Shen, J., Kim, S., & Kim, S. (2025). A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*. <https://doi.org/10.1145/3747588>
20. [20] Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., et al. (2023). *Code Llama: Open foundation models for code*. arXiv:2308.12950.
21. Lozhkov, A., Li, R., Allal, L. B., Cassano, F., Lamy-Poirier, J., Tazi, N., et al. (2024). *StarCoder 2 and The Stack v2: The next generation*. arXiv:2402.19173.
22. Sallou, J., Durieux, T., & Panichella, A. (2024). Breaking the silence: The threats of using LLMs in software engineering. In *Proceedings of the 46th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. ACM.
23. Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 754–768. IEEE.
24. [24] Perry, N., Srivastava, M., Kumar, D., & Boneh, D. (2023). Do users write more insecure code with AI assistants? In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2785–2799. ACM. <https://doi.org/10.1145/3576915.3623157>
25. Fu, Y., Liang, P., Tahir, A., Li, Z., Shahin, M., Yu, J., & Chen, J. (2025). Security weaknesses of Copilot-generated code in GitHub projects: An empirical study. *ACM Transactions on Software Engineering and Methodology*. <https://doi.org/10.1145/3716848>
26. Sandoval, G., Pearce, H., Nys, T., Karri, R., Garg, S., & Dolan-Gavitt, B. (2023). Lost at C: A user study on the security implications of large language model code assistants. In *Proceedings of the 32nd USENIX Security Symposium*.
27. Asare, O., Nagappan, M., & Asokan, N. (2023). Is GitHub's Copilot as bad as humans at introducing vulnerabilities in code? *Empirical Software Engineering*, 28(6), 129.

28. Khoury, R., Avila, A. R., Brunelle, J., & Camara, B. M. (2023). How secure is code generated by ChatGPT? In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE.
29. Siddiq, M. L., & Santos, J. C. S. (2022). SecurityEval dataset: Mining vulnerability examples to evaluate machine learning-based code generation techniques. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security (MSR4P&S)*. ACM.
30. Schreiber, M., & Tippe, P. (2025). *Security vulnerabilities in AI-generated code: A large-scale analysis of public GitHub repositories*. arXiv:2510.26103.
31. Majdinasab, V., Bishop, M. J., Rasheed, S., Moradidakhel, A., Tahir, A., & Khomh, F. (2024). Assessing the security of GitHub Copilot's generated code — A targeted replication study. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE.
32. Blyth, S., Licorish, S. A., Treude, C., & Wagner, M. (2025). *Static analysis as a feedback loop: Enhancing LLM-generated code beyond correctness*. arXiv:2508.14419.
33. Spracklen, J., Wijewickrama, R., Sakib, A. H. M. N., Maiti, A., Viswanath, B., & Jadliwala, M. (2025). We have a package for you! A comprehensive analysis of package hallucinations by code-generating LLMs. In *Proceedings of the 34th USENIX Security Symposium*.
34. Zhao, Y., Wu, M., Hu, X., & Xia, X. (2025). *HFuzzer: Testing large language models for package hallucinations via phrase-based fuzzing*. arXiv:2509.23835.
35. Torres-Arias, S., Afzali, H., Kuppusamy, T. K., Curtmola, R., & Cappos, J. (2019). in-toto: Providing farm-to-table guarantees for bits and bytes. In *Proceedings of the 28th USENIX Security Symposium*.
36. Peng, S., Kalliamvakou, E., Cihon, P., & Demirel, M. (2023). *The impact of AI on developer productivity: Evidence from GitHub Copilot*. arXiv:2302.06590.
37. Ziegler, A., Kalliamvakou, E., Li, X. A., Rice, A., Rifkin, D., Simister, S., Sittampalam, G., & Aftandilian, E. (2022). Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS)*. ACM.
38. Vaithilingam, P., Zhang, T., & Glassman, E. L. (2022). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*. ACM.
39. Imai, S. (2022). Is GitHub Copilot a substitute for human pair-programming? An empirical study. In *Proceedings of the 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. ACM.
40. Barke, S., James, M. B., & Polikarpova, N. (2023). Grounded Copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1), 85–111.
41. Prather, J., Reeves, B. N., Leinonen, J., MacNeil, S., Randrianasolo, A. S., Becker, B. A., Kimmel, B., Wright, J., & Briggs, B. (2024). The widening gap: The benefits and harms of generative AI for novice programmers. In *Proceedings of the 2024 ACM Conference on International Computing Education Research (ICER)*, 302–311. ACM.

42. Sabouri, S., Eibl, P., Zhou, X., Ziyadi, M., Medvidović, N., Lindemann, L., & Chattopadhyay, S. (2025). Trust dynamics in AI-assisted development: Definitions, factors, and implications. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 1678–1690. IEEE.
43. Parasuraman, R., & Riley, V. (1997). Humans and automation: Use, misuse, disuse, abuse. *Human Factors*, 39(2), 230–253.
44. Skitka, L. J., Mosier, K. L., & Burdick, M. (1999). Does automation bias decision-making? *International Journal of Human-Computer Studies*, 51(5), 991–1006.
45. Bainbridge, L. (1983). Ironies of automation. *Automatica*, 19(6), 775–779.
46. Carlini, N., Tramèr, F., Wallace, E., Jagielski, M., Herbert-Voss, A., Lee, K., et al. (2021). Extracting training data from large language models. In *Proceedings of the 30th USENIX Security Symposium*.
47. Carlini, N., Ippolito, D., Jagielski, M., Lee, K., Tramèr, F., & Zhang, C. (2023). Quantifying memorization across neural language models. In *International Conference on Learning Representations (ICLR)*.
48. Ciniselli, M., Pascarella, L., & Bavota, G. (2022). To what extent do deep learning-based code recommenders generate predictions by cloning code from the training set? In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*. ACM.
49. Al-Kaswan, A., Izadi, M., & van Deursen, A. (2024). Traces of memorisation in large language models for code. In *Proceedings of the 46th International Conference on Software Engineering (ICSE)*. ACM.
50. Al-Kaswan, A., & Izadi, M. (2023). The (ab)use of open source code to train large language models. In *Proceedings of the 2nd International Workshop on Natural Language-Based Software Engineering (NLBSE)*. IEEE.
51. European Parliament and Council of the European Union. (2024). *Regulation (EU) 2024/1689 laying down harmonised rules on artificial intelligence (Artificial Intelligence Act)*. Official Journal of the European Union.
52. Board of Governors of the Federal Reserve System & Office of the Comptroller of the Currency. (2011). *Supervisory Guidance on Model Risk Management*. SR Letter 11-7 / OCC Bulletin 2011-12.
53. Board of Governors of the Federal Reserve System, Office of the Comptroller of the Currency, & Federal Deposit Insurance Corporation. (2026). *Interagency Guidance on Model Risk Management*. SR Letter 26-2 / OCC Bulletin 2026-13, 17 April 2026.
54. Prudential Regulation Authority. (2023). *Model risk management principles for banks*. Supervisory Statement SS1/23, Bank of England.
55. National Institute of Standards and Technology. (2023). *Artificial Intelligence Risk Management Framework (AI RMF 1.0)*. NIST AI 100-1.
56. International Organization for Standardization. (2023). *ISO/IEC 42001:2023 — Information technology — Artificial intelligence — Management system*.
57. European Parliament and Council of the European Union. (2022). *Regulation (EU) 2022/2554 on digital operational resilience for the financial sector (Digital Operational Resilience Act)*. Official Journal of the European Union, L 333.

58. Souppaya, M., Scarfone, K., & Dodson, D. (2022). *Secure Software Development Framework (SSDF) Version 1.1*. NIST Special Publication 800-218.
59. European Banking Authority. (2025). *AI Act: Implications for the EU banking and payments sector*.
60. Basel Committee on Banking Supervision. (2013). *Principles for Effective Risk Data Aggregation and Risk Reporting (BCBS 239)*. Bank for International Settlements.
61. Basel Committee on Banking Supervision. (2021). *Principles for Operational Resilience*. Bank for International Settlements.
62. Financial Conduct Authority. (2022). *TSB fined £48.65m for operational resilience failings*. Press release, 20 December 2022.
63. McIntosh, S., Kamei, Y., Adams, B., & Hassan, A. E. (2014). The impact of code review coverage and code review participation on software quality: A case study of the Qt, VTK, and ITK projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*. ACM.
64. Bacchelli, A., & Bird, C. (2013). Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE.
65. Sadowski, C., Söderberg, E., Church, L., Sipko, M., & Bacchelli, A. (2018). Modern code review: A case study at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. ACM.
66. Boehm, B., & Basili, V. R. (2001). Software defect reduction top 10 list. *Computer*, 34(1), 135–137.
67. Leveson, N. G. (2011). *Engineering a Safer World: Systems Thinking Applied to Safety*. MIT Press.
68. Perrow, C. (1984). *Normal Accidents: Living with High-Risk Technologies*. Basic Books.
69. Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B., & Zimmermann, T. (2019). Software engineering for machine learning: A case study. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE.
70. Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., & Dennison, D. (2015). Hidden technical debt in machine learning systems. In *Advances in Neural Information Processing Systems (NeurIPS)*.