BSc Software Engineering

Bachelor Thesis

Toward Overcoming the AI Productivity Paradox

A Prescriptive Governance Framework for Maintainable AI-Assisted Coding

Pau Marro

Semester

Spring 2026

Date of Submission

11 June 2026

First Examiner

Samuel Boguslawski

Second Examiner

Adam Roe

CODE University of Applied Sciences

Table of Contents

1. Introduction **4**
2. Background **6**
2.1 AI-Assisted Coding 6
2.2 Technical Debt 6
2.3 Maintainability 7
2.4 Governance **7**
3. Methodology **8**
3.1 Phase 1: Systematic Literature Review 8
3.1.1 Search Strategy 8
3.1.2 Study Selection (Inclusion / Exclusion Criteria) 9
3.1.3 Data Extraction 11
3.1.4 Data Synthesis 11
3.2 Phase 2: Expert Interviews 12
3.3 Phase 3: Framework Synthesis 13
4. RQ1: Maintainability Risks and Technical Debt Drivers **13**
4.1 Surface Code Smells 14
4.2 Structural and Architectural Debt 15
4.3 Security-Related Technical Debt 16
4.4 Review Burden and Rejection Rates 17
4.5 Cognitive and Behavioural Drivers 18
4.6 Task and Context Sensitivity 19
4.7 Training Data and Model Selection 19
5. RQ2: Quality Assurance and Mitigation Practices **21**
5.1 Steering the Agent: Prompt and Context Engineering 21
5.2 Automated Gates and In-Loop Repair 22
5.3 Human Review, Accountability, and Skill Retention 24
5.4 Harness Engineering and its Residue 25
6. RQ3: Prescriptive Governance Framework **28**
6.1 Framework Derivation: From Map to Pillars 28
6.2 Pillar 1: Cognitive Engagement and Accountability 29
6.3 Pillar 2: Review Processes and Human-in-the-Loop 29
6.4 Pillar 3: Layered Quality Controls 30
6.5 Pillar 4: Monitoring, Metrics, and Feedback Loops 31
6.6 Context Moderators 32
7. Discussion **33**
7.1 Implications 33
7.2 Limitations 33
7.3 Threats to Validity **34**
8. Conclusion **35**
References 37
Appendix A: Database Search Strings 41
Declaration on Plagiarism and AI 42

1. Introduction

AI-assisted coding (AAC) tools have achieved substantial adoption in enterprise software development, with empirical studies reporting productivity improvements of up to 56% on isolated coding tasks [45], [18] and, at the project level, increased code contributions and broader developer participation [60], [47]. At first glance, these metrics suggest that AAC tools deliver on their promise of accelerated development velocity.

Yet emerging evidence reveals that these gains impose significant long-term maintainability costs. A difference-in-differences analysis of open-source projects found that AI-assisted contributions consistently required more rework to meet repository standards, while core developers saw a decline in original productivity as they absorbed a growing review workload [57]. A dynamic simulation model confirms an inverted-U pattern: moderate assistance improves performance, but excessive use compounds technical debt (TD) into a self-reinforcing decline in maintainability [5]. A randomised controlled trial even found AI assistance slowing experienced contributors on mature codebases, despite their belief that they had been faster [8].

This evidence exposes a fundamental paradox: AAC tools accelerate code production while simultaneously increasing TD, review burden, and maintenance costs. The critical question is therefore not whether to adopt these tools, but how to effectively govern their use so that the speed gains translate into durable progress rather than deferred rework.

This thesis investigates the AI productivity paradox through three research questions. RQ1 asks what maintainability risks and TD drivers are empirically associated with AAC in established codebases. RQ2 asks what quality assurance (QA) practices are identified as effective for mitigating these risks. RQ3 asks what prescriptive governance framework can be recommended based on the thematic synthesis of empirical findings and practitioner insights.

To answer these questions, the thesis employs a three-phase mixed-methods design: a systematic literature review (SLR) of 51 primary studies, four semi-structured expert interviews analysed via thematic analysis, and a framework synthesis integrating both streams into a four-pillar governance model. Chapter 2 provides background;

Chapter 3 details the methodology; Chapters 4 and 5 present the integrated findings for RQ1 and RQ2; Chapter 6 synthesises the governance framework for RQ3; Chapter 7 discusses implications, limitations, and threats to validity; and Chapter 8 concludes.

2. Background

2.1 AI-Assisted Coding

AI-assisted coding (AAC) refers to the use of large language model (LLM)-based tools to generate, modify, or review source code; prominent examples include GitHub Copilot, ChatGPT, Claude Code, Cursor, Devin, and Amazon Q. These tools span a spectrum of autonomy, from inline autocompletion, through chat-based assistants that generate snippets the developer copies in, to agentic tools that plan, execute, and iterate on multi-file changes inside the repository with limited per-step intervention. The thesis adopts AAC as an umbrella term but, because the SLR corpus and the interviews are dominated by chat-based and agentic usage respectively, calibrates its findings to those modes. One difference matters most for interpreting the evidence: agentic tools can execute code and iterate against feedback such as test or linter output, whereas completion and chat modes cannot. The temporal evolution of these tools is addressed in Chapter 3 and Section 7.3.

2.2 Technical Debt

Technical debt (TD) describes the accumulated cost of suboptimal design and implementation decisions that expedite short-term delivery at the expense of long-term maintainability [22]. As with financial debt, TD carries interest: every later change to affected code costs more than it would have if the debt did not exist. The interest manifests as additional bugs, longer change-cycle times, and recurring refactoring effort.

Not all technical debt is equal. Following Fowler's quadrant [22], this thesis works with the prudent-versus-reckless distinction. *Prudent debt* is taken on deliberately: it is documented, time-bound, and planned for repayment. *Reckless debt* arises from missing knowledge, weak discipline, or absent remediation plans, often without the team realising it. The axis recurs throughout the thesis, because a central concern is that AAC's speed tempts teams toward reckless debt while presenting it as progress.

AAC introduces TD drivers that differ in character from those familiar in traditional software engineering. The specific TD categories empirically associated with AAC are presented in Chapter 4.

2.3 Maintainability

Maintainability is the degree to which a software system can be modified effectively and efficiently by its maintainers [28], commonly decomposed into modularity, reusability, analysability, modifiability, and testability. *Code quality*, as used here, is narrower: the conformance of individual artefacts to design principles and style standards, covering readability, low complexity, absence of code smells [21], and test coverage, all measurable via static analysis. Maintainability is a system-level property that emerges from how those artefacts compose, so the thesis treats maintainability as the outcome variable and code quality as one observable proxy. The focus is on established, or brownfield, environments, which are mature systems with entrenched conventions, legacy constraints, and complex cross-file dependencies; greenfield projects, by contrast, offer architectural flexibility and smaller codebases better suited to the limited context windows of current models.

2.4 Governance

Governance refers to the policies, processes, and controls through which an organisation directs and constrains an activity to achieve declared objectives while managing risk. In software engineering, governance encompasses decision rights (who may approve what), process controls (how work moves from intent to production), technical controls (automated gates that enforce rules), and socio-technical controls (norms, training, and accountability structures). Governance can be descriptive (documenting existing practice) or prescriptive (recommending practices to adopt). Traditional software-engineering governance models were developed for human-authored code and have not been validated against the scale, autonomy, and non-determinism of AAC [64], [2]; whether and how they require adaptation is an open question this thesis addresses prescriptively in Chapter 6.

3. Methodology

This thesis employs a three-phase sequential mixed-methods design: a systematic literature review (Phase 1) to establish the empirical evidence base, semi-structured expert interviews (Phase 2) to ground that evidence in practitioner experience, and a thematic framework synthesis (Phase 3) that integrates both streams into the prescriptive governance framework. The sequential ordering is deliberate: each phase conditions the next, with the review shaping the interview protocol and both feeding the synthesis. The remainder of this chapter details each phase in turn.

3.1 Phase 1: Systematic Literature Review

To answer RQ1 and RQ2 (Chapter 1), I conducted a Systematic Literature Review (SLR) following the guidelines established by Kitchenham and Charters for software engineering research [31], and reported the results according to the PRISMA 2020 statement [43]. The protocol-driven procedure was chosen over a narrative review for its transparency and reproducibility, and because it limits researcher bias.

The review proceeded in three stages. Planning fixed the scope, search strategy, and selection criteria before any searching began, preventing post-hoc adjustment of the protocol to the results. Conducting covered the database search, snowballing, screening, and data extraction. Reporting synthesised the extracted evidence thematically; the answers to RQ1 and RQ2 follow in Chapters 4 and 5.

3.1.1 Search Strategy

The search targeted IEEE Xplore and the ACM Digital Library, which together index the primary venues for empirical software engineering research (ICSE, FSE, MSR, EMSE, TOSEM, and TSE). Springer-hosted venues such as EASE and SANER fell outside the database search and were recovered only partially through snowballing, a limitation revisited in Section 7.3. The temporal scope was restricted to January 2023 to March 2026: since Copilot reached general availability in June 2022 and ChatGPT launched in November 2022, earlier studies predominantly examine tool generations no longer in common use.

The search string combined three concept groups joined with AND, with synonyms within each group joined with OR; the full strings for both databases are reproduced in *Appendix A*.

The database search returned 39 IEEE Xplore hits and 92 ACM Digital Library hits before deduplication.

To complement the database search and mitigate the well-documented recall limitations of keyword retrieval in a fast-moving and terminologically inconsistent field [56], I performed forward and backward snowballing on a set of anchor references. Backward snowballing inspected the reference lists of each anchor paper and yielded 10 further studies; forward snowballing, using Google Scholar's "Cited by" feature on each anchor, contributed 17 additional studies, for a snowballing total of 27 of the 51 included studies (53%).

3.1.2 Study Selection (Inclusion / Exclusion Criteria)

Studies were screened in two stages against ten predefined criteria (IC1-IC10), summarised in Table 1. Stage 1 applied IC1-IC9 to titles and abstracts; Stage 2 applied IC10 (minimum methodological rigour) to full texts.

Table 1. Inclusion and exclusion criteria.

Code	***Criterion***	***Include***	***Exclude***
IC1	Date	Jan 2023 – Mar 2026	Earlier publications
IC2	Language	English	Non-English
IC3	Study type	Primary empirical: controlled experiments, quasi-experiments, case studies, surveys, interview studies, mixed methods, action research	Secondary studies (SLRs, mapping studies), opinion pieces, tool demos without evaluation, posters, workshop proposals
IC4	Intervention	AAC tools (Copilot, ChatGPT, Cursor, Codeium, Claude Code, …) used to write or modify source code	AI used only for testing, requirements, project management, UX, or other non-coding tasks
IC5	Outcome	Reports on code quality, maintainability, TD, review effort, defect rates, churn, readability, complexity, or governance/mitigation practices	Studies that measure only productivity, speed, or developer satisfaction without any quality dimension
IC6	Context	Software engineering settings (industry or open-source)	Non-software domains; purely pedagogical contexts
IC7	Publication type	Peer-reviewed papers, or preprints with a clearly described methodology	Blog posts, white papers, trade-magazine articles
IC8	Availability	Full text accessible	Abstract only
IC9	Duplicates	Keep the published version	Drop the preprint duplicate

Code	Criterion	Include	Exclude
IC10	Minimum rigour	Documented methodology (data collection, participants, analysis)	Claims without described evidence-gathering

Selection process. Figure 1 presents the PRISMA 2020 flow diagram [43], with separate streams for the database search and the snowballing supplement. The database search identified 131 records (92 ACM; 39 IEEE); after deduplication with Zotero, 129 unique records remained for title-and-abstract screening. Stage 1 excluded 98 records against IC1-IC9, and of the 24 full texts retrieved (7 were inaccessible) all met the minimum rigour criterion, so no records were excluded at Stage 2.

Snowballing on the anchor references contributed a further 27 studies (10 backward; 17 forward), yielding a final corpus of 51 primary studies.

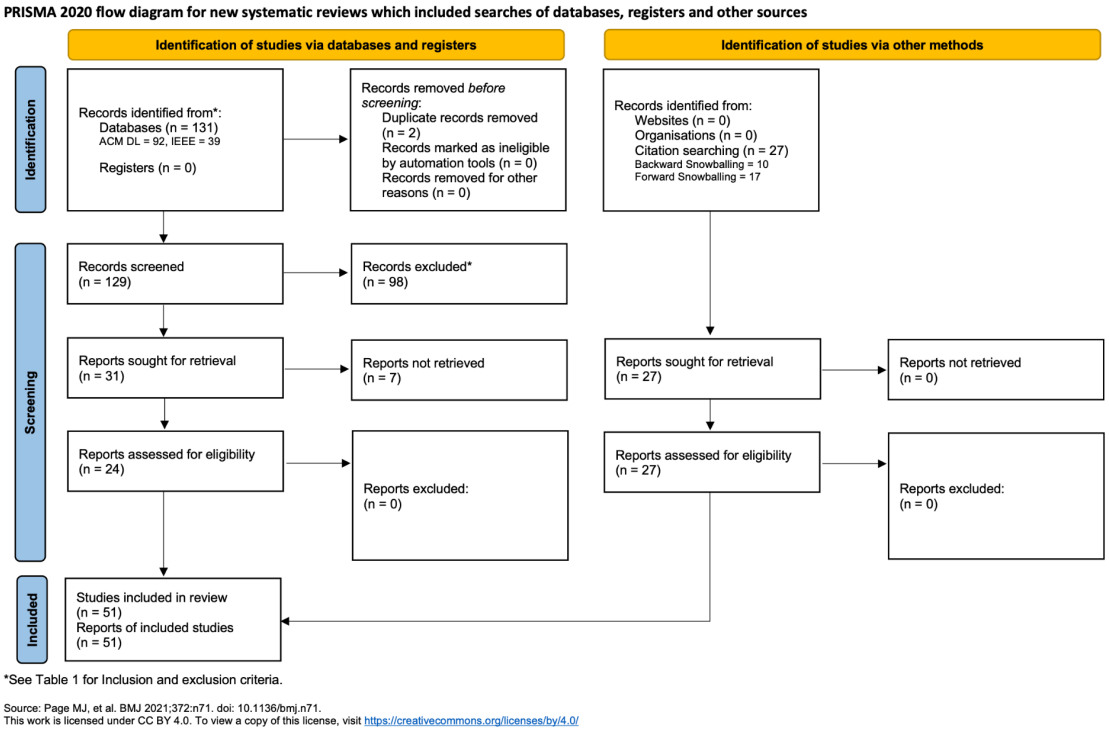

Figure 1. PRISMA 2020 flow diagram

3.1.3 Data Extraction

I developed a standardised extraction form recording, for each study:

- Bibliographic metadata.
- Study design (methodology, data source, sample size, and unit of analysis), to weigh evidence by rigour.
- AAC tool(s) studied (model family, version, and usage mode).
- Codebase context (greenfield versus established, language, and whether the evaluation used benchmarks or real repositories).
- RQ1 evidence (reported maintainability risks and TD drivers, with effect sizes where available);
- RQ2 evidence (mitigation practices, with reported effectiveness and evaluation conditions).

3.1.4 Data Synthesis

Extracted data were synthesised using thematic analysis [11]; Phases 2 and 3 (Sections 3.2 and 3.3) apply the same procedure.

For RQ1, I aggregated all reported maintainability risks across the 51 studies and grouped them via iterative open coding, yielding five recurrent categories: surface code smells; structural and architectural debt; security-related technical debt; review burden and rejection; and cognitive and behavioural drivers. Within each I distinguished convergent from single-study findings and flagged tensions where studies disagreed.

For RQ2, I extracted each reported mitigation practice and classified it by control mode (technical versus human-process) and lifecycle stage (pre-generation, in-loop, pre-merge, post-merge), recording its effectiveness and evaluation context, and mapping it back to the RQ1 risks it addresses. This also exposed a scope boundary. Most of the corpus's quantitative severity baselines come from single-shot generation: the model produces code once, without running it or reacting to feedback, as in benchmark solutions and chat-produced snippets. Such baselines cannot estimate what survives an agent's execution-and-repair loop. Its agentic studies [2], [19], [23], [24], [25], [26], [55] are largely post-merge repository mining: they show what escapes the loop in the wild, not how teams operate it. Phase 2 addresses that operational gap directly.

RQ3 is answered in Chapter 6 through the Phase 3 synthesis of the SLR and interview findings.

3.2 Phase 2: Expert Interviews

Phase 2 consisted of four semi-structured interviews with experienced software engineers, selected via purposive sampling for variation in codebase type and domain; all four maintain established codebases, though P1's lint-enforced onion/DDD harness is a greenfield build; the interviews are read with the thesis's brownfield focus in mind. The data were analysed via thematic analysis using the procedure of Section 3.1.4. Participants are referred to by anonymised identifiers (P1-P4), with company and product names replaced by generic descriptors and informed-consent records retained by the author. Table 2 profiles them; the findings appear in Chapters 4 and 5 alongside the SLR evidence. Interviews capture mitigation strategies (RQ2) well but have an inherent limit for raw risks (RQ1): practitioners rarely work in a total governance vacuum, so the qualitative RQ1 insights qualify the SLR baselines rather than replace them. The 1-5 severity figures participants assigned, reported throughout Chapter 4, are subjective judgements rather than measurements.

Table 2. Interview participant profiles.

ID	***Role / domain***	***Codebase characteristics***	***Primary AAC tool(s)***	***AI-generated share***	***AAC experience***
P1	Software & AI engineer, B2B SaaS (medical-device mfg)	Greenfield lint-enforced onion / DDD; also maintains established projects	Cursor → Claude Code (+ cloud review agents)	~90–100%	~1.5 years
P2	Founder / lead engineer, production-planning SaaS	Brownfield TypeScript monorepo, ~200k LOC	Claude Code (GCP Vertex)	>85%	~2 years
P3	Backend engineer, mobile-app SaaS	Brownfield ~5 yrs, migrated .NET → TypeScript	ChatGPT (research) Claude Code	>50%	~3 years

P4	AI engineer / informal lead, social-media analytics	Established Python Microservices ~20–30k LOC, 3.5 yrs (+ 8yr brownfield service in separate team)	Cursor (in production) Codex / Claude Code (for automation)	>40%	~3.5 years

3.3 Phase 3: Framework Synthesis

Findings from both the SLR and interviews were integrated through a unified thematic synthesis. The same thematic logic applied in Phase 2 was extended to the literature findings, producing a consolidated set of themes that form the basis of the prescriptive governance framework. The framework structure emerged from the combined analysis, moderated by context factors identified across both evidence streams.

4. RQ1: Maintainability Risks and Technical Debt Drivers

This chapter answers RQ1 by integrating the two evidence streams. Each subsection leads with the SLR evidence, adds the corroborating, qualifying, or contradicting practitioner evidence, and closes with a short synthesis. Sections 4.1-4.5 catalogue the five risk categories of Section 3.1.4, from codebase symptoms (4.1-4.4) to the cognitive drivers behind them (4.5), while Sections 4.6-4.7 analyse the factors that moderate them. Two qualifications run throughout. First, the risks are not equally tractable: surface code smells (Section 4.1) are the layer current models and tooling increasingly catch, whereas structural, security, and cognitive risks are progressively harder to detect automatically and to clear by iteration. Second, the SLR baselines largely describe ungoverned, pre-agentic usage, whereas the practitioners I interviewed work within established agentic pipelines with rigorous quality safeguards (Section 7.3).

4.1 Surface Code Smells

SLR evidence

Ungoverned AAC output carries a persistent code-smell burden even when it is functionally correct, and the effect is large and consistent. Liu et al. [35] find maintainability issues in 47% of ChatGPT-generated programs (45% even among those that pass their tests), and Paul et al. [44] measure a +63.34% code-smell density against a professional human Java baseline, with design-level smells such as Unutilized Abstraction appearing only in LLM output [1], [46], [50], [59], [63]. Two findings sharpen the concern for established codebases. First, iteration does not reliably help: Horikawa et al. [26] find 56.1% of agentic “readability” commits actually lower the Maintainability Index. Second, the gate does not reliably catch smells: across 1,210 merged agent PRs, Cynthia et al. [19] find the introduced issues are overwhelmingly smells rather than bugs (1,059 versus 48), and 53.3% of Claude Code PRs still carry a quality issue after merge, evidence that correctness-focused review lets structural defects through.

Practitioner evidence

The interviews qualify these figures along a temporal axis. P1, P3, and P4 each rated current shallow-smell prevalence at 2 on the 5-point scale (Section 3.2), well below the SLR baselines, and reported the risk attenuating with current-generation models without disappearing; P2 likewise described cosmetic smells as largely tractable, his attention having shifted to harder structural problems. The mechanisms behind this attenuation are examined in Section 5.2.

Integrated Interpretation

The SLR's high prevalence reflects unmitigated, pre-agentic baselines, while the lower severity practitioners report reflects models and gates that catch much of this layer: the residual risk is real but shrinking. It is also the first instance of a recurring pattern in which iteration clears some risks and leaves others untouched, developed in Chapter 6 into the reachable-versus-resistant distinction.

4.2 Structural and Architectural Debt

SLR evidence

Ungoverned AAC frequently produces code that is structurally shallow (functionally correct but lacking the abstraction and modularity that make later modification safe) or that does not conform to the surrounding architecture, the risk that matters most in established codebases. Molison et al. [40] capture the trade-off: LLMs generate fewer functional bugs than humans but comparable-or-worse maintainability, with more critical structural issues at higher complexity. Watanabe et al. [55] give the strongest brownfield evidence: 27.1% of revised agentic PRs needed structural refactoring to align with project architecture, and 29.0% needed documentation synchronisation. Enterprise-scale repository mining corroborates a slower, compounding effect: He et al. [24] find Cursor adoption drives a persistent rise in cognitive complexity (+41.64%, $p<0.001$), how hard code is to understand, even as the short-term velocity gain dissipates within two months, and a companion study (Agarwal et al. [2], 401 repositories) converges on a +42.9% increase with no velocity benefit.

Two mechanisms explain why. The model cannot reason about what it cannot fit in context: functional correctness collapses to 15% on methods with cross-class

dependencies, far below the rates for self-contained methods, because the interfaces of the interacting classes fall outside its context window [16]. Unable to hold the wider structure in view, it gravitates to local edits: refactoring agents bias toward low-level cosmetic changes and underperform at the architectural restructuring established codebases actually need [25], [15].

Practitioner evidence

All four ranked architectural misalignment a primary risk (P1 rated it 3.5 unmitigated) and located its cause in the codebase and on how the tool is used, rather than the model. P2 noted that agents invent new conventions wherever existing ones are not machine-detectable, and P3 that brownfield's competing patterns leave the agent imitating an arbitrary one. P4 stressed the cost of retrofitting architectural rules onto an existing repository, where one new rule can force changes across many already-passing files. The machine-readable rules they use to contain this are examined in Sections 5.2 and 5.4.

Integrated Interpretation

The SLR establishes prevalence and the mechanism, context-window limits; the interviews supply the root cause, the heterogeneous brownfield conventions on which those limits bite, and the lever, architectural rules made machine-checkable with error messages the agent can consume. This is conditionally reachable debt (Recommendation 3.5).

4.3 Security-Related Technical Debt

SLR evidence

While security is not this thesis's primary lens; security flaws enter as a maintainability liability, consuming review and remediation effort that would otherwise pay down other debt. Cotroneo et al. [17] find ungoverned AAC produces thousands more vulnerable Python and Java samples than human code, dominated by high-severity Common Weakness Enumerations; security benchmarks surveyed by Klemmer et al. [32] place 27-40% of Copilot output as vulnerable, with developers writing less secure code while believing the opposite, and AI-generated code carries about twice the rate of high-risk alerts [38].

Practitioner evidence

None of the four named security as a daily risk, though P3 blocks destructive operations with a command allowlist. That silence fits the SLR's point that security flaws accumulate undetected in AI-generated code until discovered through review or exploitation [17], [30]. Silence here is a limit of self-report rather than evidence of low risk, so the category reads as under-determined: present in the literature, latent in practice.

4.4 Review Burden and Rejection Rates

SLR evidence

AAC inflates review burden and decouples acceptance from quality. Adoption of an AI review bot raised mean PR-closure time from 5h52m to 8h20m ($p<0.001$), with 26.2% of its comments ignored [13], and agentic PRs are rejected far more often than human ones (Devin 65.03%, Claude Code 38.24%, against 17.12% for human Python PRs [23]). The deeper problem is that merging is a poor proxy for quality: 45.1% of accepted Claude Code PRs needed revision before merge and 39.9% bundled several unrelated changes into one PR, against 12.2% for humans [55], while high agentic merge rates (86.9%) bring no matching quality gain [25]. Cynthia et al. [19] explain this discrepancy, noting that most technical debt enters the codebase through pull requests that pass standard merge criteria; because reviewers focus primarily on functional correctness.

Practitioner evidence

Practitioners emphasized how AI inflates review demands: P4 noted that PR sizes have grown fivefold, while P2 observed that these larger submissions overwhelm reviewers, causing more code smells to pass unnoticed. P3 warned that without minimum-depth rules, low-tier code becomes a "vibe-code lane" where review lapses. The size caps, deterministic gates, and risk-tiered review adopted in response are presented in Chapter 5.

Integrated Interpretation

The deeper reading is that acceptance has decoupled from quality: a merged PR is weaker evidence of maintainable code than it was when humans authored at human volume. Practitioners restore the signal upstream, with pre-review gates, PR-size

discipline, and risk-tiered review, which return in Chapter 6 as Recommendations 2.2, 1.3, and 2.3.

4.5 Cognitive and Behavioural Drivers

SLR evidence

A coherent socio-technical picture links three mechanisms: automation bias, eroding accountability, and the skill loss they produce. Automation bias, the tendency to over-rely on automated output and scrutinise it less than manual work [3], [32], shows up directly: eye-tracking finds programmers give Copilot-generated code significantly less visual attention ($p<0.001$) [3], and the user studies surveyed in [32] find developers writing less secure code while reporting careful review. Tang et al. [52] document the habit of accepting AI edits without understanding them, known as vibe coding, a hidden-debt mechanism that compounds as a project grows. Alami et al. [4] argue that LLM-assisted review weakens the way peer review normally spreads responsibility for quality across a team, because developers defer to the AI's verdict without owning the judgement. Borg et al. [10] frame the cumulative result as cognitive debt: a skill erosion distinct from code-level debt, in which delegating thinking to AI gradually costs the team the design judgement and codebase familiarity that long-term maintenance needs. DePalma et al. [20] show the same gap from the output side, where ChatGPT's refactoring looks like improvement while static analysis still finds violations across every quality attribute.

Practitioner evidence

Cognitive debt was universally recognised and ranked highest by most. P1 named loss of code familiarity, complacency, and eroded design thinking as his top three risks; P3 ranked a weakened mental model and slower debugging first, drawing the line between AI-assisted coding and vibe coding at whether the developer can explain a change before accepting it. P4 confirmed automation bias and scope-creep from unrequested changes, and P2 observed that velocity tempts teams to ship code they do not understand.

Integrated Interpretation

SLR and interviews converge on cognitive debt as the most consequential AAC-specific driver, because it turns what could be prudent debt into reckless debt: a team that accepts AI output without internalising it has neither documented the

shortcut nor kept the knowledge needed to repay it. This is the foundation of Pillar 1 in Chapter 6.

4.6 Task and Context Sensitivity

SLR evidence

AAC quality is highly conditional on task and context. Gains concentrate on boilerplate and documentation; complex legacy work yields little benefit and often negative ROI [61]. Where the developer lacks the domain knowledge to judge AI output, Mendes et al. [39] go further and recommend disabling AAC, finding that forced reliance in low-context phases adds more debt than it removes. Established codebases are precisely where these conditions bite, since their cross-file dependencies, legacy conventions, and proprietary APIs intensify every risk above.

Practitioner evidence

All four named the codebase as the dominant moderator. P3 called brownfield-with-mixed-conventions the worst case but noted, counter-intuitively, that agents aid maintainability when used to interrogate commit history rather than to author. P1 described greenfield-by-design, shaping the codebase to suit the agent from day one, which is infeasible in brownfield. P2 reported a sharp split between his well-supported TypeScript stack (low risk) and his Python/C++ optimization code (high risk).

Integrated Interpretation

The SLR locates context sensitivity methodologically and the interviews confirm it as the dominant moderator of every risk above; because these factors decide which recommendations apply, they return as the Context Moderators of Section 6.6.

4.7 Training Data and Model Selection

SLR evidence.

A final class of debt originates upstream, in the training data and the choice of model. Smells are largely inherited rather than filtered: over 85% of those in the CodeSearchNet-Python corpus propagate into output, and scaling parameters from 0.5B to 7B does not reduce them [53], [58], while Improta et al. [27] show active amplification, with maintainability smells redistributing to 65.37% of generated code against roughly 20% in training data. The implication is that the pipeline is a lever,

not a constant: cleaning training data halves low-quality functions without hurting correctness [27], and fine-tuning on cleaned data cuts generates smells by 61-83% [58]. Model choice is itself measurable, with generated-Python smell prevalence ranging from 43.8% to 68.8% across models [48], so teams that self-host gain a lever most AAC consumers lack.

Practitioner evidence.

None of the four train or fine-tune their own models, so training-data hygiene is outside their control. Their one upstream lever is model selection, where all four weighed context-window capacity explicitly, since it bounds how much cross-file and architectural scope the agent can reason about (Section 4.2).

Integrated Interpretation.

Training-data hygiene establishes upstream causality but is actionable only for organisations that run their own pipelines, so it is scoped out of the core framework; model selection, including reasoning capabilities and context-window capacity, remains a lever every team controls.

5. RQ2: Quality Assurance and Mitigation Practices

This chapter answers RQ2 by integrating the Phase 1 and Phase 2 evidence in the same format as Chapter 4. Every mitigation in the corpus resolves into one of two control modes. Technical controls, that are enforced deterministically by tooling (linters, type-checkers, tests, architectural fitness functions, and the repair loops built on them), which an agentic loop can therefore run against itself; and human-process controls (review, sign-off, ownership, and the shared artefacts a team maintains), which cannot be reduced to a deterministic check. Plotting the reported practices against these two modes and the lifecycle stages at which they act (pre-generation, in-loop, pre-merge, post-merge) provides the map in Table 3. The chapter walks that map: the guides that steer the agent before generation (5.1), the in-loop gates and repair that constrain it (5.2), the human controls of review, accountability, and skill retention (5.3), and the harness into which teams consolidate those guides and gates (5.4). The chapter closes by reading the completed map; Chapter 6 turns that diagnosis into prescription.

5.1 Steering the Agent: Prompt and Context Engineering

SLR evidence

The cheapest place to intervene is before generation, by engineering what the agent sees. Summary-mediated prompts reach 93% sub-task success, though they force the developer to understand the task first [52]; supplying a purpose-written codebase summary rather than the existing README raises function-test pass rates roughly sevenfold on average [6], and naming the smells to avoid cuts their prevalence by 55.8% [53]. These levers carry a debt class of their own in undocumented prompts and unvalidated context. Watanabe et al. [55] push them into durable form with a dedicated agent-instruction file (e.g. CLAUDE.md) of formatting, design, and architectural rules, and Yu [61] calls for retrieval-augmented, project-level context injection to counter the limits behind architectural drift.

Practitioner evidence

All practitioners agreed on the decisive variable is timing, not volume: the failure mode, as P1 puts it, is the wrong context rather than too little of it. P1 inserts a planning phase before any code is generated, in which the agent critiques its own plan against the repository's conventions, so context is applied at the moment of

design rather than dumped upfront. P2 works the same principle from the consumption side, having the agent audit the relevant feature first so the window fills with what the task touches, and budgeting tokens deliberately because model attention measurably decays as the window grows.

Integrated Interpretation

The two streams describe different layers of one intervention. The SLR mostly evidences prompt engineering, while the practitioners moved up to context engineering, through just-in-time provisioning, audit-first priming, and token budgets. Most of this remains per-session effort, the instruction file of [55] being the corpus's one step toward durability; Section 5.4 shows how teams convert them into versioned infrastructure that persists across sessions.

5.2 Automated Gates and In-Loop Repair

SLR evidence

Where review is the human control, static-analysis gates are its automated counterpart. Combined tool portfolios surface over 60% of maintainability issues, though none alone suffices [35]; specialised tools extend their reach, with Semgrep at 97.75% true positives on security patterns [27] and SonarQube carrying the same gates into continuous pipelines [7]. Two limits bound what gates achieve. First, conventional analysis cannot see architectural drift or cross-file inconsistency, the risks that matter most in established codebases [58]; catching those takes architectural fitness functions (dependency-cruiser, pytest-archon) that a team must configure, and an LLM judging its own smells is only partly reliable (F1 0.85-0.89 for structural smells but below 0.40 for semantic ones [49]). Second, a gate's value depends on what is done with its output: feeding findings back as the next prompt turns detection into repair, fixing 20-60% of issues per round [35], and naming the smell lifts the fix rate from 34.4% to 87.1% [62]. The gains decay fast, with returns diminishing past two or three rounds and multi-round fixing tending to preserve or raise complexity [14], [36]; open-ended re-prompting can even add defects, because the model cannot see downstream callers [42] and re-expresses logic rather than simplifying it [26]. Separating generation from critique across two models [54] and anchoring repair to deterministic rules, which cuts the failure rate from 52.6% to 41% [41], address this. Repair, in short, should be same-turn and rule-anchored.

Practitioner evidence

For practitioners, gates serve as a mandatory precondition rather than an after-the-fact validation, with successful teams embedding repair directly into the generation loop to automate compliance. P2 enforces deterministic type and lint gates that every change must pass before he reviews anything, to the point that authorship "stops mattering": once the checks pass, the standard is implemented. P1 encodes his onion/DDD architecture as lint rules whose error messages are written for the agent to read, so it sees the violation and self-corrects within the same turn; P3 likewise wires lint and format hooks into the agent loop so errors are fed back inside the turn rather than later in CI. None of this is automatic out of the box, though: the agent self-repairs only against gates the team deliberately wires in, and all practitioners make conventions machine-detectable precisely so the agent cannot invent its own.

Integrated Interpretation

The SLR establishes coverage; the interviews show what changes when gate output becomes agent input rather than a CI failure triaged later. That change rests on two properties the "same-turn versus iterative" framing conflates: feedback determinacy, where a machine-checkable oracle narrows the model's search space (the mechanism behind the named-smell and rule-anchored gains above), and loop closure, where re-applying the same oracle on every pass keeps the target from drifting and ends repair only when the rule is satisfied. Together they dissolve the SLR's warning that multi-turn re-prompting amplifies debt: it holds for open-ended feedback, not for a loop closed against a precise check. All participants use gates to supply the oracle via lint-enforced architectures and in-loop hooks to close the loop; P1's architectural-misalignment severity falling from 3.5 to 1 shows the payoff. Chapter 6 (Recommendation 3.2) accordingly recommends determinate, loop-closed repair over open-ended re-prompting.

5.3 Human Review, Accountability, and Skill Retention

SLR evidence

Human review is the most endorsed control: every study retains it. Klemmer et al. [32] propose three-step validation (manual inspection, copy-execute-fix, peer review with tests), and Alami et al. [4] insist peer review stay the primary accountability mechanism, with LLMs confined to first-pass screening. Industry mirrors this: Google requires code-owner sign-off on every LLM-generated change [64], and Atlassian uses a staged human-in-the-loop (HITL) model [51]. The corpus also records the control's failure mode: Kashif et al. [29] reach 91% functional correctness in a governed HITL workflow while TD volume persists, which is why every workflow above layers review on top of automated gates rather than substituting one for the other. For skill retention the corpus offers prescriptions, not evaluations: active oversight rather than passive acceptance [20], mentorship kept inside peer review [4] and AI-literacy training [32], [39].

Practitioner evidence.

All four retain human review, but each locates it differently. P3 still reads every line and treats AI-authored code exactly as human-authored, drawing the line at vibe coding; P1 distributes review across a multi-stage agentic loop, in which the agent drafts a plan, critiques it against the repository's conventions (persona files acting as critics), executes, and re-checks; P4 runs an automated AI review on every diff as a first-pass screen without ceding final judgement. P3 also tiers review intensity by risk, with full scrutiny for critical code and a lighter pass for internal tooling, naming the accountability cost of the lighter tier. P1 adds a loss no gate registers: even where review survives as a quality gate, it no longer serves as the team's skill-transmission mechanism, which is why Pillar 1 treats skill retention as a separate control.

Integrated Interpretation.

Both streams agree review is the highest-leverage human control, but the interviews show it is not staying where the SLR places it. Review is migrating rather than disappearing: upstream into the agentic loop, where P1's plan critique acts before any code exists; downstream into narrowed human attention on contracts and architecture, exemplified by P2 reviewing whether his machine-readable contracts changed; and across P3's risk-tiered gradient of scrutiny. What the literature treats as a single pre-merge gate, practice has decomposed into a distribution of attention

across the lifecycle. The refinements this migration yields, the mental-model acceptance test, the multi-stage loop, and tiered intensity with an explicit floor, become Recommendations 1.2, 2.1, and 2.3.

5.4 Harness Engineering and its Residue

Two of the three mitigation clusters in this chapter, the guides that steer the agent (5.1) and the gates that check it (5.2), can in practice be considered a single act of engineering, and practitioners have begun to name it. Böckeler [9] defines *the harness* as everything in an agentic system except the model: the guides that steer the agent beforehand (instruction files, conventions, skills) and the sensors that let it self-correct afterward (linters and other static-analysis tools, test suites, architectural fitness functions). P1 used it unprompted throughout his interview, and P4 defined harness engineering as "the discipline of designing the systems, boundaries, sandboxes, and feedback loops that wrap around a raw AI model to turn it into a reliable, production-grade software engineer". Both practitioners agreed that the discipline becomes integral to an engineer's job as the share of manually written code shrinks.

Its recurring components span advisory and deterministic layers: machine-readable conventions, domain models, and persona files that critique plans (P1); per-domain context loaded on demand, with Zod schemas and an ISA-95 model as contracts (P2); and a determinacy ranking from advisory files through deterministic hooks to irreversible-action permission boundaries (P3). The harness turns guides, gates, and repair into infrastructure rather than per-session effort, while carrying debt of its own in drifting artefacts (P4), which Recommendation 3.3 places under versioned management. The SLR never names the practice; its support arrives only in fragments [55], [6], [61], [62]. That fragmentation is itself a finding: practice has integrated, under one name, controls the literature evaluates in isolation, so reported per-component effect sizes never measure the composed system teams actually run. Crucially, the harness binds only the machine-checkable controls; the architectural judgement, accountability, and cognitive engagement it cannot encode remain the engineers to enforce. Plotting the mitigation practices among these two control modes (technical and human-process) revealed an insightful discovery about the agent-loop-resistant residue.

Table 3. Complete inventory of the mitigation practices extracted from both evidence streams, classified by the Section 3.1.4 scheme of control mode against lifecycle stage ([n] = SLR study; Pn = interview participant).

Lifecycle stage	***Technical***	***Human-process***
Pre-generation	Codebase-summary / few-shot context [6]; versioned harness, context and machine-readable contract files - ISA-95, CLAUDE.md, skills (P1-P4); command allowlist (P3); training-data cleaning [27], [58]	Comprehension-forcing and smell-listing prompt protocols [52], [53]; panel-of-experts plan critique (P1); audit-first context priming (P2); model-selection testing (all P)
In-loop	In-turn lint/format hooks (P3); agent-readable architectural lint (P1); architectural fitness functions [58]; static-analysis-guided and dual-model repair [41], [54], [33]; named-smell repair prompts [62]; LLM smell-detection [49]	Multi-stage agentic review loop: plan → critique → execute → re-check (P1)
Pre-merge	Linter portfolios [35]; Semgrep [27]; pre-commit ruff + test gates (P4); SonarQube gate [7] type-check + lint CI gate (P2); severity-triaged cloud review agents (P1)	Three-step validation [32]; code-owner sign-off [64]; peer review as distributed accountability [4]; read-every-line ownership (P3); risk-tiered review depth (P3); mental-model acceptance test (P3); staged HITL [51]
Post-merge	SonarQube continuous evaluation [7]; overnight Codex janitor (P4); debt-log garbage-collection agent (P1)	— none

The table maps controls that act on a specific change at a specific lifecycle stage; cross-cutting socio-technical practices such as training, accountability norms, and knowledge-sharing forums are continuous rather than stage-bound and are treated in Pillars 1 and 4.

The technical column is dense and well evidenced; the human-process column is thinner and dominated by review. Beyond review, neither the literature nor the interviews offered a worked-out practice for keeping human judgement sharp or guarding against over-trust. This is the chapter's sharpest finding: the cognitive and accountability risks of Section 4.5 are the ones tooling cannot absorb, yet the ones for which no worked-out mitigation yet exists. The engineer remains the maintainer

of record, but the discipline has not yet built the human practices that role demands; Chapter 6 turns this diagnosis into the framework, giving that residue explicit attention while flagging the thinly evidenced recommendations as practice-led (Section 7.1).

6. RQ3: Prescriptive Governance Framework

This chapter answers RQ3 with a four-pillar prescriptive governance framework built on the evidence of Chapters 4 and 5. Section 6.1 derives its structure from Table 3; Sections 6.2 to 6.5 present the pillars, each grouping recommendations by the maintainability concern they address and cross-referencing the supporting evidence; and Section 6.6 sets out how six context factors moderate its application. Recommendations rest on convergent evidence where the table provides it; where it is thin they are practice-led and flagged for validation. Each is stated concretely enough to be falsifiable by future empirical work.

6.1 Framework Derivation: From Map to Pillars

The framework targets established, brownfield codebases (Section 2.3), and its organising principle is where governance effort, weighed against the evidence, returns the most maintainability. Two findings set the order. First, the reachable surface risks (code smells, style, deterministic security patterns) are increasingly caught by the tools themselves. Wiring the harness that catches them remains high-return engineering; only the marginal value of further surface checks, once that harness exists, is falling. Second, the risks that resist tooling (architectural judgement, accountability, cognitive engagement) are both the most damaging in established codebases and the least addressed by existing practice. The framework's distinctive contribution is concentrated there.

The four pillars follow this reasoning. Pillar 1 (Cognitive Engagement and Accountability) leads because it governs the risks every interview named as most persistent and existing practice covers least. Pillar 2 (Review Processes and Human-in-the-Loop) and Pillar 3 (Layered Quality Controls) then structure the human judgement and the technical harness that each catch what the other cannot, and Pillar 4 (Monitoring, Metrics, and Feedback Loops) sustains both over time. This ordering marks where the framework adds what practice still lacks; it is not a ranking of value, since the harness remains the high-return foundation the later pillars assume.

6.2 Pillar 1: Cognitive Engagement and Accountability

This pillar addresses the cognitive and behavioural risks that persist regardless of model capability improvements (Section 4.5).

Recommendation 1.1: Define explicit accountability ownership for all AI-authored code. Every AI-generated change must have a named human owner who has reviewed and understood it before merge, countering the accountability erosion of Section 4.5 ([4]; all four participants).

Recommendation 1.2: Enforce a mental-model test for acceptance. Developers should be able to explain the rationale and implications of an AI-generated change before approving it. This is P3's criterion separating AI-assisted coding from vibe coding, and a direct counter to the automation bias documented via eye-tracking (Section 4.5 [3]). The underlying principle is wise delegation: never automate the engineering thinking itself, only augment it.

Recommendation 1.3: Implement PR-size discipline as an accountability lever. Cap PR size or mandate split-PR rules to prevent the review collapse documented in Section 4.4 (P4's fivefold PR growth; 39.9% of agentic PRs bundling multiple purposes [55]).

Recommendation 1.4: Maintain deliberate hand-coding zones, skill-retention practices, and training in disciplined AI use. Borg et al.'s [10] cognitive-debt construct grounds intentional non-AI coding as capability preservation against the loss of design thinking P1 identified (Section 4.5); the same study links training in targeted, critically reviewed AI use to higher-quality downstream codebases.

6.3 Pillar 2: Review Processes and Human-in-the-Loop

This pillar structures the HITL review sequence.

Recommendation 2.1: Codify a multi-stage review loop as the baseline pattern. Mandate plan review and post-execution review stages, following P1's plan-critique-execute-re-check workflow (Section 5.3) and Atlassian's staged HITL model [51]; no study in the corpus argues for autonomous merge. This preserves the plan-before-implement discipline that AI speed otherwise tempts teams to bypass.

Recommendation 2.2: Require deterministic gate passage before human review begins (P2, Section 5.2), so that reviewer attention is spent on semantic and architectural concerns rather than the surface issues static analysis already catches. This sequencing is practice-led (P2) and flagged for validation.

Recommendation 2.3: Implement risk-tiered review intensity with an explicit floor. P3's split between critical product code and internal tooling is a pragmatic governance model, but each tier needs a mandated minimum review depth to prevent the vibe-code lane of Section 4.4. This tiering is practice-led (P3) and flagged for validation.

Recommendation 2.4: Deploy AI-assisted review as a complement, never a replacement, for human review. P4's automated AI review on every diff is useful first-pass screening, but AI review cannot substitute for human architectural judgement [4], [13].

6.4 Pillar 3: Layered Quality Controls

This pillar implements deterministic, machine-readable quality gates optimised for agent consumption, targeting reachable AAC-specific debt such as the code inflation of agentic output [26] and the architectural drift of Section 4.2.

Recommendation 3.1: Deploy multi-tool static-analysis portfolios as mandatory gates. No single tool provides full coverage [35]; combine language-appropriate tools (PMD+CheckStyle for Java; Pylint+Flake8+Semgrep for Python; ESLint+Prettier for TypeScript) in pre-commit and CI pipelines.

Recommendation 3.2: Integrate static-analysis gates into the agent loop. Rather than treating lint as a CI-only check, feed errors back to the agent for in-turn self-repair with verbose, agent-parseable error messages (P1, P3; Section 5.2). Scope the repair loop to deterministic, locally resolvable rules, not structural refactors needing project-level visibility [42], [49]; deployed at ByteDance scale, this gated pattern resolved approximately 85% of targeted lint violations before human review [34].

Recommendation 3.3: Maintain version-controlled context artefacts as first-class assets. CLAUDE.md, harness, and skills files, together with architecture decision records, are governance artefacts (Section 5.4): stale or undocumented prompt-level artefacts are themselves technical debt that can actively mislead the agent (P4). This

recommendation is practice-led; the SLR supports only its components in isolation [55], [6], [61], [62] rather than as a unified versioned harness.

Recommendation 3.4: Implement safety controls and guardrails. Use command allowlists that block destructive operations (P3) and, more broadly, an AI gateway layer between developer and model to validate inputs, guard against prompt injection, and prevent leakage of proprietary code into public LLMs [30].

Recommendation 3.5: Enforce architectural constraints via machine-readable rules. Lint-enforced dependency directions (P1), abstract-base-class architecture (P4), and canonical data models (P2's ISA-95) let agents produce architecturally conforming code by default rather than drifting toward monolithic structures (Section 4.2 [2]).

6.5 Pillar 4: Monitoring, Metrics, and Feedback Loops

This pillar addresses the monitoring gap identified across all four interviews and recommended by the SLR. It is particularly important given the model-level risks of silent updates and unclear evaluation metrics.

Recommendation 4.1: Track lint-warning count trends as a leading indicator of agent-quality decay. P2 identified the lint-warning context-dilution feedback loop, where accumulated warnings inflate the context window and degrade future agent performance; a zero-warning policy or warning budget prevents this self-reinforcing cycle [37].

Recommendation 4.2: Monitor post-commit instability windows. Mao et al.'s [38] finding of 30 to 90 day post-commit instability motivates extended review windows that target abstraction and deduplication, not just functional bugs.

Recommendation 4.3: Deploy scheduled agent janitors for continuous cleanup. P4's overnight Codex agent, producing daily auto-PRs for trivial fixes, demonstrates a lightweight mechanism for preventing TD accumulation in the otherwise-empty post-merge row of Table 3. This mechanism is practice-led (P4) and awaits empirical validation.

Recommendation 4.4: Establish knowledge-sharing forums and organisational governance. P3's weekly AI meeting and P4's call for cross-team AI-strategy alignment represent socio-technical feedback loops that distribute effective practices

and surface emerging risks. Organisational governance, including clear ownership of AI policy, adversarial testing protocols, and cost management, is essential for preventing systemic debt. These socio-technical loops are practice-led and flagged for validation.

Recommendation 4.5: Implement sync-check automation for context-artefact drift and model-behaviour monitoring. Automated verification that CLAUDE.md, harness, and skills files remain synchronised with the codebase keeps stale governance artefacts from misleading agents (Recommendation 3.3). Teams should also monitor for silent model updates by tracking output-consistency metrics [2]; Azanza et al.'s [7] continuous-evaluation framework, which re-scores LLM output against a weighted rubric on each model update, offers a concrete, tool-agnostic instrument for this monitoring and a template for the empirical validation this framework calls for.

6.6 Context Moderators

The framework's application is moderated by six factors identified across both evidence streams, each shifting which recommendations carry the most weight. (1) Greenfield versus brownfield: greenfield enables architecture-by-design and cheap lint enforcement from day one, while brownfield raises the retrofit cost of Recommendation 3.5 and makes context engineering (3.3) the first investment [6], [16]. (2) Model generation: newer models shrink the shallow-smell layer, shifting effort from Pillar 3's surface gates toward architectural and cognitive controls; SLR figures from 2023-2024 likely overstate severity for current tools (Section 7.3). (3) Language and stack: strictly typed monorepos make deterministic gates (2.2, 3.1) cheap and reliable, whereas C and C++ carry the highest risk (more security findings and lower semantic correctness than Python or Java [30]) and dynamically typed languages such as JavaScript also warrant deeper review tiers [44], [48]. (4) Risk tier of the artefact: low-stakes internal tooling invites accountability erosion, raising the weight of the review floor in Recommendation 2.3 [4]. (5) Tool autonomy mode: in-loop gating (3.2) is only possible in agentic use; autocompletion and chat modes shift the burden to pre-merge review (Pillar 2; Section 2.1). (6) Token economy: heavy harnesses are token-expensive, so harness scope (3.3) must be balanced against oversight value and cost [63].

7. Discussion

7.1 Implications

For practitioners, the most actionable finding is that deterministic, agent-in-loop quality gates (Pillar 3) combined with accountability structures (Pillar 1) offer the highest leverage. Surface-level code quality is increasingly manageable through tool-model co-evolution, while cognitive and architectural risks intensify with deeper delegation [2], [10], so governance effort should shift toward the latter.

A second implication is that speed without planning creates reckless debt rather than strategic progress [40], discovered only when the system has become too fragile to change. Requirements analysis, architectural design, and deliberate planning remain as necessary as before agentic coding; Recommendations 2.1 and 3.5 operationalise that continuity.

For researchers, the temporal sensitivity of findings matters methodologically: AAC tools evolve so quickly that empirical results have a shorter shelf-life than is usual in software engineering. The gap between the SLR baselines and 2026 practitioner experience (Section 7.3) argues for longitudinal designs and explicit temporal framing in future work.

7.2 Limitations

The SLR corpus is influenced by open-source models, which underperform compared to the frontier models most developer teams actually use, and the database scope (IEEE Xplore and ACM Digital Library) may miss relevant venues, though snowballing partially compensates. Rapid model evolution further erodes comparability across studies evaluating different tool generations.

The interview sample (N=4) is small and skewed toward Claude Code and Cursor users, mostly small teams working in TypeScript and Python; no large-enterprise or regulated-industry voice is represented. Self-report bias is present as well, with no triangulation against repository metrics.

Three further limitations concern the analysis and its output rather than the evidence base itself. Because all screening, extraction, and thematic coding were performed by a single researcher, inter-rater reliability could not be established across either the

SLR or the interviews. The interviews are also cross-sectional, capturing practitioner judgement at one point in a fast-moving field rather than tracking how it shifts. Most importantly, the framework itself has not yet been validated: it is a synthesis-derived design artefact whose individual recommendations are evidence-based, but whose application as an integrated whole remains empirically untested. A 9,000-word scope compounds these constraints, bounding reporting depth and favouring breadth over exhaustive treatment of any one risk.

7.3 Threats to Validity

Construct validity is threatened by the heterogeneous operationalisation of maintainability across studies, with some using static-analysis proxies, others review outcomes, and few true long-term measurements. Internal validity is limited by the observational nature of most SLR studies and the self-reported interview data. A specific confound affects every cross-stream comparison in Chapter 4: the SLR effect sizes describe largely ungoverned AAC, whereas all four practitioners operate heavily governed pipelines, so when they rate a risk below the SLR baseline that gap cannot be attributed cleanly to newer models, model improvement and mitigation effort being inseparable in this design. The temporal-attenuation interpretation of Chapter 4 is therefore a practitioner attribution rather than an established causal finding. External validity is constrained by the benchmark dominance in the SLR and the small, homogeneous interview sample.

8. Conclusion

This thesis investigated the AI productivity paradox: AI-assisted coding tools accelerate code production while increasing technical debt, review burden, and maintenance costs. Through a systematic review of 51 empirical studies and four expert interviews, it addressed three research questions and produced a prescriptive governance framework for maintainable AI-assisted coding.

RQ1 identified five categories of maintainability risk: surface code smells, structural and architectural debt, security-related TD, review-burden inflation, and cognitive and behavioural drivers, including automation bias and vibe coding. The evidence is strongest for code-smell prevalence and the cognitive-bias mechanism; moderate to high for architectural debt and review-burden inflation, where acceptance has decoupled from quality; and, for security TD, present in the literature but latent in practice. Beneath these risks runs a recurring pattern: implementing before planning produces reckless rather than strategic debt.

RQ2 identified three clusters of effective QA practice: context and prompt engineering, automated gates with in-loop repair, and human review with preserved accountability. Teams consolidate the first two, the machine-checkable guides and gates, into a versioned harness that makes those controls durable rather than per-session effort; the accountability and judgement of the third are the residue it cannot encode. Training-data hygiene also works [27], [58], but only teams that control their own model pipelines can act on it. The decisive finding is that static analysis is necessary but insufficient: the architectural drift and cross-file inconsistency that threaten established codebases most become reachable only where teams deliberately configure architectural rules, and the cognitive risks remain beyond tooling altogether.

RQ3 synthesised these clusters into the four-pillar framework of Chapter 6: cognitive engagement and accountability, review processes and human-in-the-loop, layered quality controls, and monitoring with feedback loops. The framework derives from the diagnostic map of mitigation evidence.

Its application is moderated by codebase maturity, model generation, language and stack, artefact risk tier, tool autonomy mode, and token economy. Governance works on two fronts: building the harness that turns gateable risks into gated ones, and

managing the residue the loop cannot reach. It is on the second front, where tooling cannot absorb the work, that this framework stakes its distinctive claim.

Future work should pursue longitudinal TD measurement of AAC-touched files over 12 or more months, large-enterprise validation of the governance framework, empirical evaluation of the proposed monitoring metrics, and investigation of cross-team AI-strategy governance at the organisational level.

References

[1] Abbassi, A. A., Da Silva, L., Nikanjam, A., & Khomh, F. (2025). A taxonomy of inefficiencies in LLM-generated Python code. arXiv:2503.06327.

[2] Agarwal, S., He, H., & Vasilescu, B. (2026). AI IDEs or autonomous agents? Measuring the impact of coding agents on software development. In Proc. MSR ’26. ACM.

[3] Al Madi, N. (2023). How readable is model-generated code? Examining readability and visual inspection of GitHub Copilot. In Proc. ASE ’22. IEEE/ACM.

[4] Alami, A., Jensen, V., & Ernst, N. (2025). Accountability in code review: The role of intrinsic drivers and the impact of LLMs. ACM Transactions on Software Engineering and Methodology, 34(8), Article 233.

[5] Anderson, E. G., Parker, G. G., & Tan, B. (2025). Beyond productivity: Evaluating the hidden costs of generative AI in software development. SSRN Working Paper No. 5842302.

[6] Arun, et al. (2025). LLMs for generation of architectural components: An exploratory empirical study in the serverless world. arXiv:2502.02539.

[7] Azanza, M., et al. (2025). Tracking the moving target: A framework for continuous evaluation of LLM test generation in industry. In Proc. EASE ’25. ACM.

[8] Becker, J., Rush, N., Barnes, E., & Rein, D. (2025). Measuring the impact of early-2025 AI on experienced open-source developer productivity. arXiv:2507.09089.

[9] Böckeler, B. (2026). Harness engineering for coding agent users. martinfowler.com. https://martinfowler.com/articles/harness-engineering.html

[10] Borg, M., Hewett, D., Hagatulah, N., Couderc, N., Söderberg, E., Graham, D., Kini, U., & Farley, D. (2026). Echoes of AI: Investigating the downstream effects of AI assistants on software maintainability. arXiv:2507.00788 (Empirical Software Engineering, in submission).

[11] Braun, V., & Clarke, V. (2006). Using thematic analysis in psychology. Qualitative Research in Psychology, 3(2), 77-101.

[12] Chen, T. (2024). The impact of AI-pair programmers on code quality and developer satisfaction: Evidence from TiMi studio. In Proc. GAIIS 2024. ACM.

[13] Cihan, et al. (2025). Automated code review in practice. In Proc. ICSE-SEIP 2025. IEEE/ACM.

[14] Clark, A., Igbokwe, D., Ross, S., & Zibran, M. F. (2024). A quantitative analysis of quality and consistency in AI-generated code. IEEE (MSR 2024 ecosystem).

[15] Cordeiro, J., Noei, S., & Zou, Y. (2026). An empirical study on the code refactoring capability of large language models. ACM Transactions on Software Engineering and Methodology.

[16] Corso, V., Mariani, L., Micucci, D., & Riganelli, O. (2024). Assessing AI-based code assistants in method generation tasks. In Proc. ICSE-Companion ’24. IEEE/ACM.

[17] Cotroneo, D., Improta, C., & Liguori, P. (2025). Human-written vs. AI-generated code: A large-scale study of defects, vulnerabilities, and complexity. arXiv:2508.21634.

[18] Cui, K. Z., Demirer, M., Jaffe, S., Musolff, L., Peng, S., & Salz, T. (2024). The productivity effects of generative AI: Evidence from a field experiment with GitHub Copilot. MIT Working Paper.

[19] Cynthia, S. T., Muttakin, A., & Roy, B. (2026). Beyond bug fixes: An empirical investigation of post-merge code quality issues in agent-generated pull requests. In Proc. MSR ’26. ACM.

[20] DePalma, K., Miminoshvili, I., Henselder, C., Moss, K., & AlOmar, E. A. (2024). Exploring ChatGPT’s code refactoring capabilities: An empirical study. Stevens Institute of Technology (preprint).

[21] Fowler, M. (1999). Refactoring: Improving the design of existing code. Addison-Wesley Professional.

[22] Fowler, M. (2009). TechnicalDebtQuadrant. martinfowler.com. https://martinfowler.com/bliki/TechnicalDebtQuadrant.html

[23] Gonzalez, R., & El Mezouar, M. (2025). Different by design: Understanding human-AI collaboration through GitHub pull requests. In Proc. CASCON 2025. IEEE.

[24] He, H., Miller, C., Agarwal, S., Kästner, C., & Vasilescu, B. (2026). Speed at the cost of quality: How Cursor AI increases short-term velocity and long-term complexity in open-source projects. In Proc. MSR ’26. ACM.

[25] Horikawa, K., Li, H., Kashiwa, Y., Adams, B., Iida, H., & Hassan, A. E. (2025). Agentic refactoring: An empirical study of AI coding agents. arXiv:2511.04824.

[26] Horikawa, K., Horikawa, K., Kashiwa, Y., Uwano, H., & Iida, H. (2026). Do AI agents really improve code readability? In Proc. MSR ’26. ACM.

[27] Improta, C., et al. (2025). Quality in, quality out: Investigating training data’s role in AI code generation. In Proc. ICPC 2025. IEEE/ACM.

[28] ISO/IEC 25010:2011. Systems and software engineering - Systems and software quality requirements and evaluation (SQuaRE) - System and software quality models. Geneva: ISO.

[29] Kashif, S. M., Li, R., Liang, P., Tahir, A., Feng, Q., Li, Z., & Shahin, M. (2026). Beyond functional correctness: Design issues in AI IDE-generated large-scale projects. ACM Transactions on Software Engineering and Methodology. arXiv:2604.06373.

[30] Kharma, M. F., Choi, S., Alkhanafseh, M., & Mohaisen, D. (2026). Security and quality in LLM-generated code: A multi-language, multi-model analysis. arXiv:2502.01853 (IEEE Transactions on Dependable and Secure Computing, in submission).

[31] Kitchenham, B., & Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering (Technical Report EBSE-2007-01). Keele University & Durham University.

[32] Klemmer, J. H., Horstmann, S. A., Patnaik, N., Ludden, C., Burton, C., Jr., Powers, C., Massacci, F., Rahman, A., Votipka, D., Lipford, H. R., Rashid, A., Naiakshina, A., & Fahl, S. (2024). Using AI assistants in software development: A qualitative study on security practices and concerns. In Proc. CCS ’24. ACM.

[33] Kouemo Ngassom, S., Moradi Dakhel, A., Tambon, F., & Khomh, F. (2024). Chain of targeted verification questions to improve the reliability of code generated by LLMs. In Proc. AIware ’24. ACM.

[34] Li, Y., Xie, L., Chen, Y., Long, Q., He, X., Duan, W., Yao, Z., Geng, L., Xu, J., & Han, X. (2025). BitsAI-Fix: LLM-driven approach for automated lint error resolution in practice. arXiv:2508.03487.

[35] Liu, Y., Le-Cong, T., Widyasari, R., Tantithamthavorn, C., Li, L., Le, X.-B. D., & Lo, D. (2024). Refining ChatGPT-generated code: Characterizing and mitigating code quality issues. ACM Transactions on Software Engineering and Methodology, 33(5), Article 116.

[36] Liu, Z., Tang, Y., Luo, X., Zhou, Y., & Zhang, L. F. (2024). No need to lift a finger anymore? Assessing the quality of code generation by ChatGPT. arXiv:2308.04838 (IEEE TASE).

[37] Liu, Y., Widyasari, R., Zhao, Y., Irsan, I. C., & Lo, D. (2026). Debt behind the AI boom: A large-scale empirical study of AI-generated code in the wild. arXiv:2603.28592.

[38] Mao, T., Zhao, D., Tang, H., Wang, X., & Zhang, H. (2026). A large-scale empirical study of AI-generated code in real-world repositories. arXiv:2603.27130 (ACM, in submission).

[39] Mendes, W., Souza, S., & de Souza, C. R. B. (2024). "You're on a bicycle with a little motor": Benefits and challenges of using AI code assistants. In Proc. CHASE '24. IEEE/ACM.

[40] Molison, A. S., Moraes, M., Melo, G., Santos, F., & Assunção, W. K. G. (2025). Is LLM-generated code more maintainable & reliable than human-written code? arXiv:2508.00700 (IEEE).

[41] Moussa, S., & Olsson, M. (2025). Integrating static analysis and LLMs to refactor complex methods (Bachelor thesis, DT133G). Mid Sweden University.

[42] Nunes, H., Figueiredo, E., Rocha, L., Nadi, S., Ferreira, F., & Esteves, G. (2025). Evaluating the effectiveness of LLMs in fixing maintainability issues in real-world projects. (IEEE/ACM, 2025).

[43] Page, M. J., McKenzie, J. E., Bossuyt, P. M., et al. (2021). The PRISMA 2020 statement: An updated guideline for reporting systematic reviews. BMJ, 372, n71.

[44] Paul, D. G., Zhu, H., & Bayley, I. (2025). Investigating the smells of LLM-generated code. arXiv:2510.03029 (extended version of ICCBDCS 2025). Oxford Brookes University.

[45] Peng, S., Kalliamvakou, E., Cihon, P., & Demirer, M. (2023). The impact of AI on developer productivity: Evidence from GitHub Copilot. arXiv:2302.06590.

[46] Rabbi, M. F., Champa, A. I., Zibran, M. F., & Islam, M. R. (2024). AI writes, we analyze: The ChatGPT Python code saga. In Proc. MSR '24. IEEE/ACM.

[47] Song, L., et al. (2024). The impact of generative AI on collaborative open-source software development: Evidence from GitHub Copilot.

[48] Sousa, E., Santos, E., Sampaio, A., & Bezerra, C. (2025). Comparing structural quality of code generated by LLMs: A static analysis of code smells. Universidade Federal do Ceará.

[49] Souza, S., Santana, A., Figueiredo, E., Muzetti, I., Montandon, J. E., & Briand, L. (2026). Beyond strict rules: Assessing the effectiveness of large language models for code smell detection. IEEE Transactions on Software Engineering. arXiv:2601.09873.

[50] Sun, X., Ståhl, D., Sandahl, K., & Kessler, C. (2026). Quality assurance of LLM-generated code: Addressing non-functional quality characteristics. Journal of Systems and Software, 238, Article 112885.

[51] Takerngsaksiri, W., Tantithamthavorn, C., Fu, M., Pasuksmit, J., Chen, K., & Wu, M. (2025). Code readability in the age of large language models: An industrial case study from Atlassian. arXiv:2501.11264.

[52] Tang, N., Smith, E., Huang, Y., McMillan, C., & Li, T. J.-J. (2025). Exploring direct instruction and summary-mediated prompting in LLM-assisted code modification. arXiv:2508.01523.

[53] Velasco, A., Rodriguez-Cardenas, D., Khati, D., Palacio, D. N., Alif, L. R., & Poshyvanyk, D. (2026). A causal perspective on measuring, explaining and mitigating smells in LLM-generated code. In Proc. ICSE ’26. IEEE/ACM.

[54] Wadhwa, N., Pradhan, J., Sonwane, A., Sahu, S. P., Natarajan, N., Kanade, A., Parthasarathy, S., & Rajamani, S. (2024). CORE: Resolving code quality issues using LLMs. Proc. ACM on Software Engineering (FSE), 1, Article 36.

[55] Watanabe, K., et al. (2026). On the use of agentic coding: An empirical study of pull requests on GitHub. ACM Transactions on Software Engineering and Methodology.

[56] Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. In Proc. EASE 2014, pp. 1-10. ACM.

[57] Xu, F., Medappa, P. K., Tunç, M. M., Vroegindeweij, M., & Fransoo, J. C. (2025). AI-assisted programming decreases the productivity of experienced developers by increasing the technical debt and maintenance burden. arXiv:2510.10165.

[58] Xue, Z., Zhang, X., Gao, Z., Hu, X., Gao, S., Xia, X., & Li, S. (2026). Clean code, better models: Enhancing LLM performance with smell-cleaned dataset. ACM Transactions on Software Engineering and Methodology.

[59] Yetiştiren, B., Özsoy, I., Ayerdem, M., & Tüzün, E. (2023). Evaluating the code quality of AI-assisted code generation tools: An empirical study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. arXiv:2304.10778.

[60] Yeverechyahu, U., Medappa, P. K., & Tunç, M. M. (2024). The impact of large language models on open-source innovation: Evidence from GitHub Copilot. SSRN Electronic Journal.

[61] Yu, L. (2025). Paradigm shift on coding productivity using GenAI. In Proc. EASE ’25. ACM.

[62] Zhang, B., Liang, P., Feng, Q., Fu, Y., & Li, Z. (2024). Copilot-in-the-loop: Fixing code smells in Copilot-generated Python code using Copilot. In Proc. ASE ’24. IEEE/ACM.

[63] Zhong, S., Zou, Y., & Adams, B. (2025). Developer-LLM conversations: An empirical study of interactions and generated code quality. arXiv:2509.10402.

[64] Ziftci, C., Nikolov, S., Sjövall, A., Kim, B., Codecasa, D., & Kim, M. (2025). Migrating code at scale with LLMs at Google. In Proc. FSE Companion ’25. ACM.

Appendix A: Database Search Strings

IEEE Xplore

```
("AI-assisted coding" OR "AI-assisted programming" OR "AI-generated code"
OR "LLM-generated code" OR "GitHub Copilot" OR "code generation"
OR "large language model")
AND
("technical debt" OR "code quality" OR "maintainability" OR "code smell"
OR "software maintenance" OR "rework" OR "defect" OR "code review")
```

ACM Digital Library

```
[[All: "ai-assisted coding"] OR [All: "ai-generated code"]
OR [All: "llm-generated code"] OR [All: "github copilot"]
OR [All: "large language model" AND "code"]]
AND
[[All: "technical debt"] OR [All: "code quality"]
OR [All: "maintainability"] OR [All: "software maintenance"]]
```

Declaration on Plagiarism and AI

I hereby confirm that I have written the thesis titled "*Toward Overcoming the AI Productivity Paradox: A Prescriptive Governance Framework for Maintainable AI-Assisted Coding*" by myself, without contributions from any sources other than those cited in the text and bibliography. All graphics, drawings and images not created by me but included in this thesis, along with any uses of generative AI have been fully and accurately referenced.

Furthermore, I confirm that neither this work nor parts of it have been previously or concurrently used as an assessment submission in other courses or in other examination proceedings.

In the table below, I list the generative AI tools used in creating my thesis, as well as a description of the purpose and extent of the use of each tool

AI Tool Used	**Purpose of use and description of extent of use**
Claude Code	Literature and interview data organization assistance. As well as assistance with thematic analysis and extraction.
Claude/ChatGPT (Langdock)	Drafting, editing support and language refinement.

All AI-generated content has been critically reviewed, verified against primary sources, and substantially revised by me. The intellectual contributions, research design, analysis, and conclusions are entirely my own work.

Location, Date	**Student Signature**
Berlin, Germany, 10.06.2026	