PROCESS DISCIPLINE AS THE KEY VARIABLE IN AI-ASSISTED ENTERPRISE SOFTWARE DEVELOPMENT: A NATURAL EXPERIMENT

PREPRINT

Adam Zachary Wasserman
CTO, Buckler
adam.wasserman@buckler.ai

March 2026

ABSTRACT

Over 12 months (March 2025–March 2026), a two-person team using AI coding assistants produced 2.6 million lines of code across 2,929 commits and retained approximately 250,000 lines after disciplined refactoring: a 90% elimination rate. Every line that survived was reviewed, evaluated against enterprise requirements, and kept or discarded by a human architect. GitClear (2025, 211M lines across Google, Microsoft, and Meta) reports that industry cleanup activity collapsed from 25% to under 10% after AI adoption. This team sustained 90%. The dataset spans two process conditions applied to the same team using the same tools.

Under a structured AI-specific SDLC: 1,484,038 lines produced, 811,684 lines eliminated (55% cleanup ratio), 3,977 function points delivered, 18 of 18 enterprise audit dimensions satisfied, 68 test files, 13 CI/CD pipelines. Without it: 1,153,069 lines produced, 483,991 lines eliminated (42% cleanup ratio), 428 functional function points (FP) delivered (working demo with live prospect accounts), 2 of 18 dimensions satisfied (including plain text password storage), zero test files, zero pipelines, four repositories abandoned. The unstructured condition generated code 4.2x faster by volume (360,334 lines/month vs. 84,802) and produced working software that cannot be sold to a regulated customer and must be discarded rather than remediated: zero enterprise-ready function points. The structured condition produced at 9–11x the published elite productivity benchmark (227 FP/FTE-month vs. 20–26 for best-in-class manual teams; Capers Jones/SPR, 26,000+ projects).

The structured application is not a toy or prototype. At 248,710 lines of production code across 657 files, 3,977 function points, 5-service Docker orchestration, per-request cryptographic authentication, and regulatory compliance for Canadian wealth management (NI 31-103, CFR), it falls in Capers Jones's "medium-to-large" category (1,000–10,000 FP), well past the COCOMO II threshold where coordination costs dominate and productivity typically drops. No published AI productivity study has measured output at this scale and complexity; prior controlled studies used isolated tasks (Peng et al.) or single-issue contributions (METR).

This paper presents a natural experiment isolating three multiplicative conditions as the variable: structured process discipline, elite-level architectural expertise, and a clean starting codebase. Under the structured SDLC (specifications before code, behavioral tests before implementation, real-time architectural review of AI output, 13 automated quality pipelines), the team built an enterprise platform in Python/FastAPI. When control of the SDLC shifted to non-technical stakeholders, the same team produced a React application with no tests, no CI/CD, and no code review.

The technology divergence is itself a finding: without architectural constraints, the AI assistants defaulted to React, the framework most represented in their training data, and produced code of middling quality. This is not an artifact of usage or a criticism of the tools. It is a mathematical consequence of gradient descent: the optimization process converges toward the mean of the training distribution, making the most probable output the most average output. Average code does not satisfy enterprise audit requirements; those requirements exist precisely to exclude the median.

A second natural experiment reveals a more specific finding. A single developer's commit history on one codebase captures three phases: structured manual development without AI, AI-assisted development within the existing manual process, and AI-assisted development under the AI-specific SDLC. The manual process was disciplined (100% human-written code establishing architectural patterns, pull request workflow, named branches, ERD generation) but insufficient for AI-generated code: the AI produced output that passed the developer's existing quality checks while accumulating structural debt invisible to manual review. When AI-specific constraints were introduced, 175,881 lines of accumulated bloat were eliminated in a single month. This suggests that structured development processes designed for human programmers do not transfer to AI-assisted development without modification.

The published literature produces three categories of finding. Speed studies are contradictory: 55.8% speedups (Peng et al. 2023) and 19% slowdowns (METR 2025). Quality studies are less ambiguous: 4x code duplication, 60% less cleanup (GitClear 2025, 211M lines), 9% increase in bug rates with a 7.2% delivery stability decrease per 25% AI adoption increase (DORA 2024–2025). Generation loss studies document a third, less widely recognized phenomenon: AI output degrades cumulatively when fed back into AI systems. Shumailov et al. (2024, *Nature*) established the theoretical mechanism (model collapse). Shukla et al. (2025, IEEE-ISTAS) measured the rate in code: 37.6% increase in critical vulnerabilities after just 5 iterations of AI "improvement." Five independent research groups across Nature, ICLR, IEEE, and ACL confirm the same directional finding. Generation loss has direct implications for any codebase where AI assists in ongoing development rather than one-off generation: without continuous human correction, each iteration of AI-generated code moves further from the original architectural intent.

No published study has identified the variable that reconciles these three bodies of evidence. This paper provides project-level evidence that three multiplicative conditions constitute that variable: an AI-specific development process, elite-level architectural expertise, and a clean starting codebase that sets the quality ceiling for AI-generated output. The third condition is a direct response to generation loss: the starting codebase determines the context the AI absorbs, and therefore the trajectory of its output. This is consistent with DORA's (2025) characterization of AI as an "amplifier" of existing process quality.

The primary motivation for this study is to contribute empirical evidence to a body of knowledge that currently relies on aggregate survey data and controlled experiments on isolated tasks, neither of which captures AI-assisted development at enterprise scale and complexity. All metrics are derived from git commit history, source code analysis, and published industry benchmarks.

Keywords AI-assisted software development, software development lifecycle, process discipline, natural experiment, enterprise software quality, function point analysis

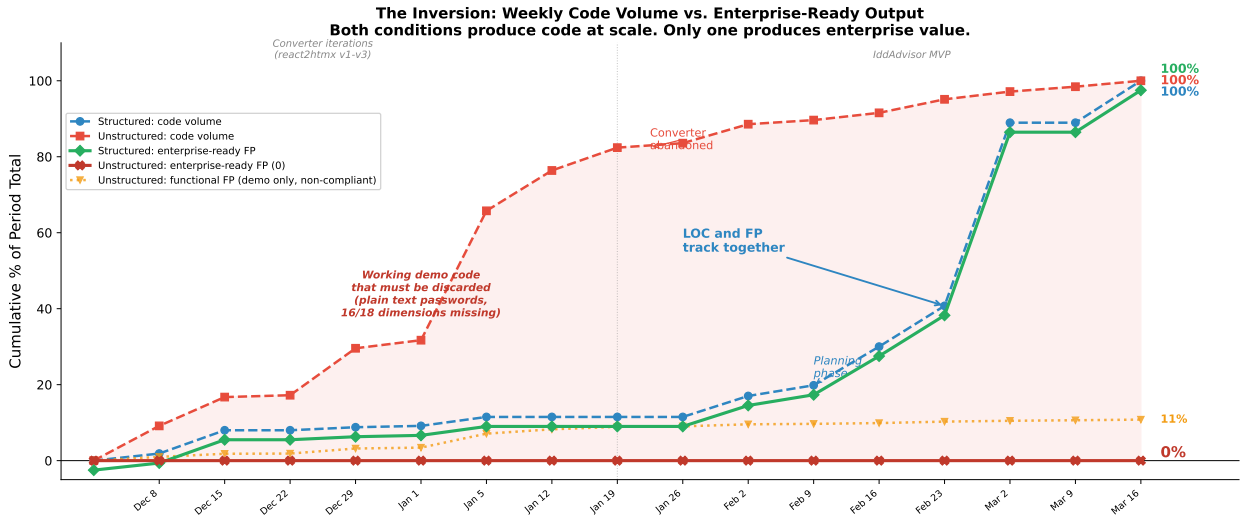


Figure 1: Weekly code volume vs. enterprise-ready output, December 2025–March 2026. Both conditions produce code at scale (dashed lines). The structured condition's enterprise-ready FP (green) tracks its code volume. The unstructured condition produces functional demo code (orange, 428 FP with live prospect accounts) but zero enterprise-ready FP (dark red): plain text password storage, 16 of 18 audit dimensions missing. The functional code must be discarded, not remediated. Same team, same AI tools, different process.[2]

1 Introduction

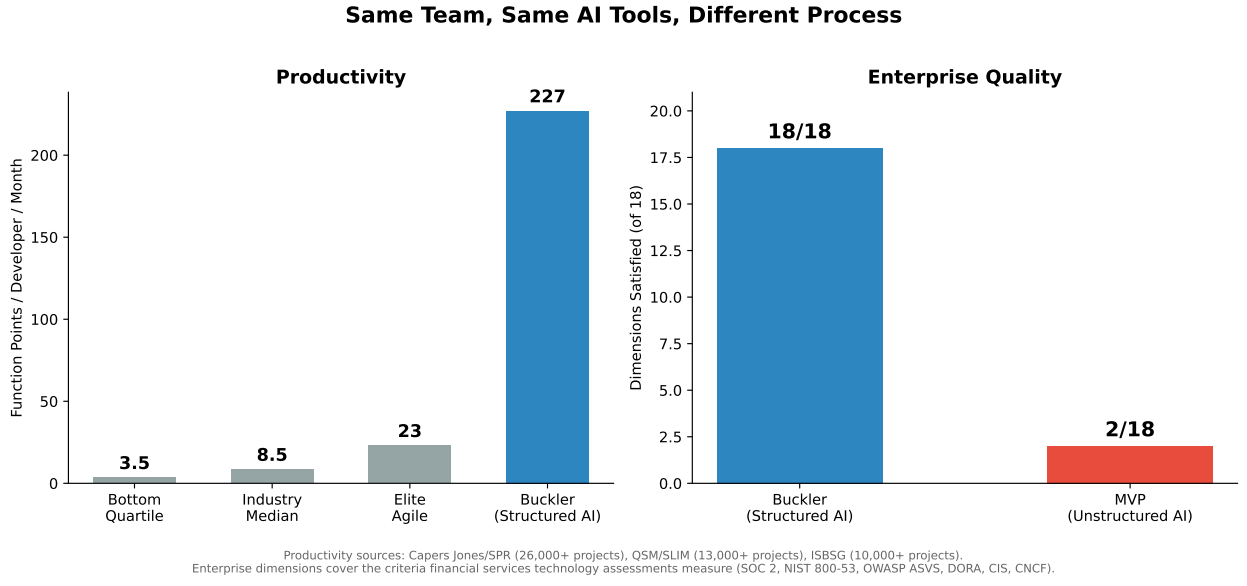


Figure 2: Software Productivity: Industry Benchmarks vs. Structured AI-Assisted Development

[2]Every data point in this figure is derived from git commit timestamps and line counts. No interpolation, smoothing, or selective filtering was applied. The only transformation is normalization to cumulative percentage of each condition's period total, applied identically to both conditions. The underlying data is available in Appendix A.

AI coding assistants have been widely available since late 2022. The question of whether they make developers faster, slower, or simply different is not settled. The published evidence is contradictory, and the contradictions are informative.

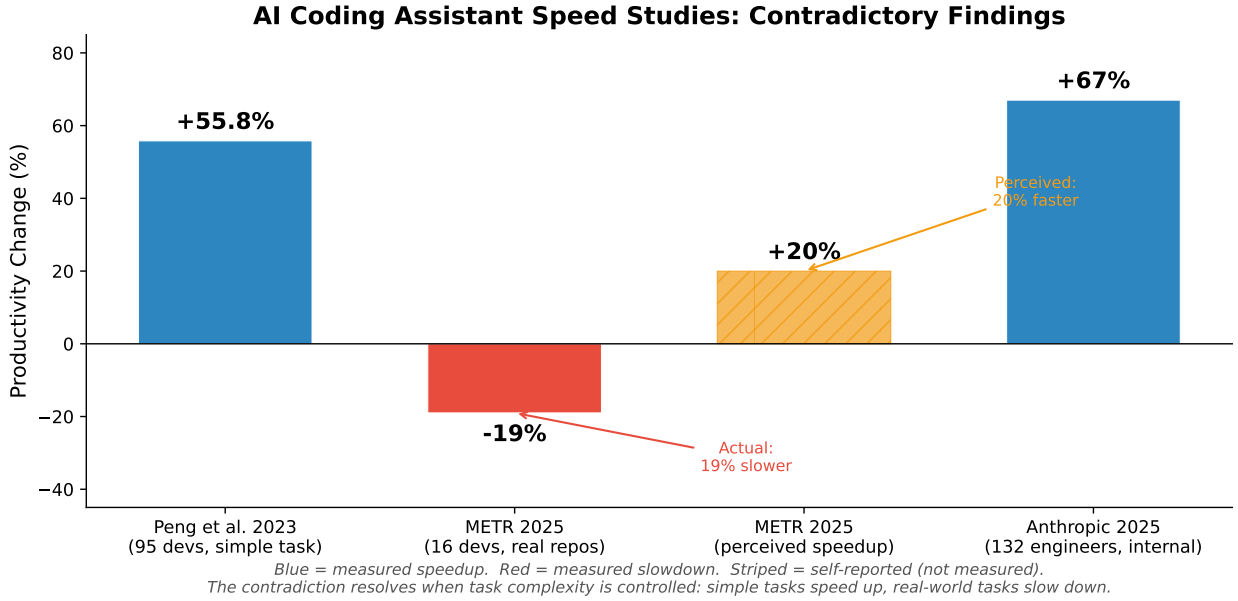


Figure 3: Published AI coding assistant speed studies produce contradictory findings. Peng et al. (2023): RCT, 95 developers, simple HTTP server task. METR (2025): 16 experienced open-source developers (avg 22K+ stars), real repository tasks. Anthropic (2025–2026): 132 self-selected engineers, internal study.

The quality data is less ambiguous (Figure 4).

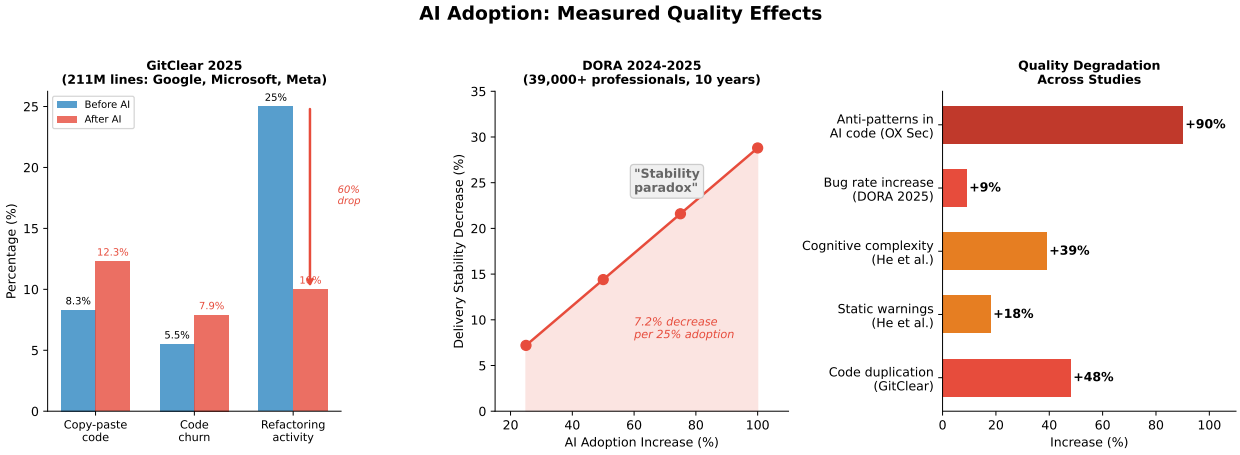


Figure 4: AI adoption quality effects. Left: GitClear (2024–2025, 211M lines across Google, Microsoft, Meta). Center: DORA (2024, 39,000+ professionals, 10 years; 2025 "stability paradox"). Right: cross-study summary including He et al. (2025, 807 repos) and OX Security (2025, 300+ repos).

A separate body of research documents iterative degradation when AI-generated code is fed back into AI systems (Figure 5).

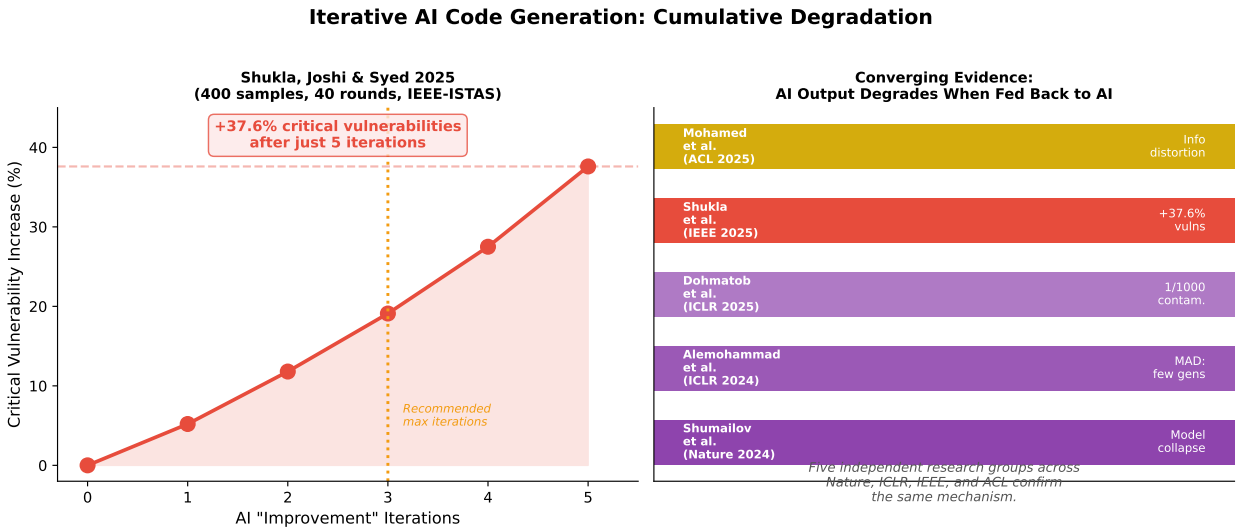


Figure 5: Left: Shukla, Joshi & Syed (2025, IEEE-ISTAS): 37.6% increase in critical vulnerabilities after 5 iterations of AI "improving" its own code. Right: five independent research groups across Nature, ICLR, IEEE, and ACL confirm that AI output degrades when fed back to AI systems.

No peer-reviewed study has isolated any variables that reconcile the speed and quality findings. The study that would settle the question requires four experimental arms: AI with structured process, AI without structured process, manual coding with structured process, manual coding without structured process. That study does not exist. What does exist is converging observational evidence that AI assistance is a multiplier of whatever process discipline surrounds it. DORA's 2025 report characterizes AI as an "amplifier" of existing organizational maturity, but this finding is based on survey-level correlational data, not controlled observation.

This paper presents a natural experiment that provides project-level evidence for the amplifier thesis. A two-person team produced two applications under different process conditions. The team and the tools were constant. The process authority changed. The outcomes diverged on 16 of 18 measurable enterprise dimensions.

The findings support five conclusions:

First, AI coding assistants produce real acceleration under both structured and unstructured conditions; the unstructured condition generated code 4.2x faster by volume than the structured condition, confirming that the tools work.

Second, code generation speed and enterprise-ready output are inversely correlated when process discipline is absent: the faster condition produced 428 functional FP (a working demo with live prospect accounts) but zero enterprise-ready FP, including plain text password storage that alone disqualifies the codebase from any financial services assessment.

Third, the process variable is not "structured vs. unstructured" in the generic sense. A second natural experiment within the same dataset shows that a structured manual development process (one that produced 100% human-written foundational code establishing the quality ceiling for the codebase) was insufficient for AI-generated code. The AI produced output that passed the developer's existing quality checks while accumulating structural debt invisible to manual review. Only the introduction of AI-specific constraints (specification constraints on AI output, real-time architectural review, BDD-first development, automated quality gates) produced the quality transition. This suggests that the process discipline literature's focus on "structured vs. unstructured" may be too coarse; the relevant variable is whether the process was designed for the specific failure modes of AI-generated code.

Fourth, the expertise required to operate an AI-specific SDLC is qualitatively different from the expertise required to write code. The architect must monitor AI output in real time, catch architectural drift before it compounds, and review through targeted questioning and selective sampling. These are observable, measurable activities, but they require the ability to recognize good architecture on sight, which is a function of experience building and maintaining enterprise systems. The METR study's finding that experienced open-source developers were 19% slower with AI is consistent: coding experience is necessary but not sufficient; architectural judgment is the additional requirement.

AI coding assistants optimize toward the mean of their training distribution (a mathematical consequence of gradient descent), so producing elite results requires continuously counteracting the tool's statistical pull toward average output. Without expert correction at every step, the output converges on the median, which by definition does not satisfy enterprise audit requirements.

The widespread expectation that AI coding assistants will enable non-experts to produce production-quality software is not supported by this data. The structured condition required elite architectural expertise applied continuously during generation; the unstructured condition, operated by the same skilled developers without that structure, produced zero enterprise-ready function points. The constraint is not coding ability but architectural judgment, which AI assistance does not replace and may obscure by producing output that appears functional while lacking systemic properties.

Fifth, AI will not produce code better than the codebase it operates in. Generation loss research (Shumailov et al. 2024, *Nature*; Shukla et al. 2025, IEEE-ISTAS) demonstrates that AI output degrades cumulatively when fed back into AI systems: 37.6% increase in critical vulnerabilities after just 5 iterations. Applied to code generation: an AI assistant operating in a well-designed codebase absorbs its conventions and produces code that follows them; an AI operating in a degraded codebase absorbs and reproduces the degradation. The starting codebase sets the quality ceiling. Application A was designed with all 18 enterprise dimensions from the beginning, providing a context that constrained the AI's output toward compliance. Application B's codebase provided no such context, and the AI reproduced its absence.

1.1 Contributions to the Body of Knowledge

This paper addresses six specific gaps in the published literature on AI-assisted software development:

1. **The process isolation gap.** No published study holds team and tools constant while varying the development process and measuring enterprise-quality outcomes. DORA (2025) characterizes AI as an "amplifier" but relies on survey-level correlational data across 39,000 respondents. He et al. (2025) compared AI-adopting vs. non-adopting repositories but did not control for process. This paper provides the first project-level observational evidence with the process variable isolated.

2. **The AI-specific process distinction.** The literature treats process discipline as binary: structured vs. unstructured. The three-phase IAM finding (Experiment 2) demonstrates a finer granularity: a disciplined manual development process, one that produced 100% human-written foundational code establishing the quality ceiling for the codebase, was insufficient for AI-generated code. No prior study has reported this distinction. The relevant variable is not process discipline in general but process discipline designed for the specific failure modes of AI-generated code.

3. **Enterprise-scale measurement.** Prior controlled studies measured simple tasks (Peng et al., an HTTP server) or isolated contributions (METR, single issues in open-source repos). No study has measured AI-assisted productivity on a medium-to-large enterprise application. At 3,977 function points, 248,710 lines of production code, and regulatory compliance requirements, Application A falls in Capers Jones's medium-to-large category, well past the COCOMO II threshold where coordination costs dominate. The published productivity benchmarks (Jones/SPR, QSM/SLIM, ISBSG) were calibrated for manual development; this is the first data point for AI-assisted development at comparable scale.

4. **The speed-quality inversion at project level.** GitClear and DORA document the negative correlation between AI adoption and code quality at aggregate level. This paper provides the first project-level demonstration with both conditions observable in the same team, using the same tools, during an overlapping time period.

5. **Generation loss in production code.** Shumailov et al. (2024, *Nature*) established the theory of model collapse. Shukla et al. (2025, IEEE-ISTAS) measured the rate in controlled experiments. No study has documented the operational cost of counteracting generation loss in a real production codebase. The 90% elimination rate (2.6 million lines produced, 250,000 retained) is that cost, measured over 12 months.

6. **The expertise requirement characterized.** The METR study found that experienced open-source developers were 19% slower with AI but offered no explanation. This paper characterizes the specific expertise required: real-time architectural monitoring during generation, targeted questioning and selective sampling during review, and continuous correction against gradient descent's pull toward the training distribution mean. These are observable, measurable activities distinct from coding experience.

2 Related Work

2.1 Speed Studies

The controlled speed studies measure different things and reach different conclusions. Peng et al. (2023, arXiv:2302.06590) tested a simple, isolated task (an HTTP server) in a peer-reviewed RCT with 95 developers and found a 55.8% speedup. METR (2025) tested real-world tasks in real repositories with 16 experienced open-source developers (average 22K+ stars) and found a 19% slowdown with a perceived 20% speedup. Anthropic's internal study (2025-2026) reported a 67% increase in merged PRs/engineer/day among 132 self-selected engineers at an AI company. The reconciliation is straightforward: AI accelerates simple, isolated tasks. On complex, real-world tasks in existing codebases, the acceleration disappears or reverses.

2.2 Quality Studies

Quality-focused studies consistently report negative downstream effects. GitClear (2024-2025, 211M lines across Google, Microsoft, and Meta) found duplicated code increased 4x, cleanup activity collapsed from 25% to under 10%, and code churn increased from 5.5% to 7.9%. Google DORA (2024) measured a 7.2% delivery stability decrease per 25% AI adoption increase. The 2025 DORA report found the "stability paradox": individual throughput up, organizational delivery metrics flat, bug rates up 9%. Xie et al. (2025, arXiv:2510.10165) found a 19% drop in original code productivity in open-source projects after Copilot introduction. OX Security (2025) identified 10 critical anti-patterns in 80-100% of AI-generated code across 300+ repositories. Stack Overflow's 2025 survey (65,000+ respondents) found positive AI sentiment dropped from 70%+ to 60%, with 66% citing "almost right but not quite" and 45% reporting that debugging AI code is more time-consuming.

He et al. (2025, arXiv:2511.04427) used a difference-in-differences design across 807 Cursor-adopting repositories matched against 1,380 controls and found "substantial but transient velocity gains alongside persistent increases in technical debt," with static analysis warnings up 18% and cognitive complexity up 39%.

2.3 Generation Loss / Iterative Degradation

Shumailov et al. (2024, *Nature* 631, 755-759) demonstrated "model collapse": irreversible quality degradation when generative models train on their own output. Alemohammad et al. (2024, ICLR) coined "Model Autophagy Disorder" (MAD) and showed appreciable quality loss in a handful of generations. Dohmatob et al. (2025, ICLR) proved that even 1-in-1,000 contamination triggers collapse, with larger models amplifying rather than mitigating the effect. Shukla, Joshi, and Syed (2025, IEEE-ISTAS, arXiv:2506.11022) conducted the first controlled study of iterative AI code generation: 37.6% increase in critical security vulnerabilities after 5 iterations, across 400 code samples and 40 rounds. Mohamed et al. (2025, ACL, arXiv:2502.20258) documented the same phenomenon in text.

2.4 The Process Variable Gap

No published study uses a controlled comparison controlling for process discipline as an independent variable alongside AI tool usage. DORA's 2025 "AI Capabilities Model" identifies 7 organizational practices that moderate AI outcomes and characterizes AI as an "amplifier" of existing maturity, but this is correlational survey data. The DeputyDev longitudinal study (2025, arXiv:2509.19708, 300 engineers, 1 year) measured AI-assisted development within a structured platform but had no "unstructured" comparison arm. The CodeScene/Borg et al. RCT (2025, arXiv:2507.00788, 151 developers) controlled for AI usage but not for process discipline. He et al. (2025) compared AI-adopting vs. non-adopting repositories but did not control for whether adopting teams had structured processes.

The gap is specific: no study holds team and tools constant while varying the development process, and measures enterprise-quality outcomes. This paper addresses that gap with observational data from a natural experiment.

3 Context and Methodology

3.1 Team and Tools

The team consisted of two developers working on enterprise financial software (Canadian wealth management, regulatory compliance):

- **Developer A** (CTO): 205 unique active days across all repositories. Works 7 days per week (a schedule sustained throughout the measurement period), which compresses calendar time relative to standard FTE assumptions. Designed the structured SDLC, the architectural framework, and served as architectural reviewer for all AI-generated code. Non-coding responsibilities include architecture, framework development, and participation in company management.
- **Developer B** (developer): 163 unique active days. Sole contributor to one major codebase (324 commits), responsible for 88% of a critical two-month infrastructure sprint (212 of 240 commits) on another. Approximately 20–30% of working time is allocated to DevOps responsibilities (deployment, infrastructure maintenance, environment configuration).

FTE calculation: Developer A = 205 developer-days. Developer B = 163 developer-days. Total: 368 FTE-days, or ~17.5 FTE-months (at 21 working days/month). This calculation uses the standard 5-day-week FTE denominator. Developer A's 7-day work week means the 205 active days were compressed into approximately 29 calendar weeks rather than the 41 weeks a 5-day schedule would require. The FTE-month figure therefore understates the team's calendar-time efficiency by approximately 30%, but is used here because published benchmarks assume 5-day work weeks.

AI coding assistants were introduced in approximately March 2025. No AI tools were used by any team member prior to that date; all code written before March 2025 is 100% human-generated. From March 2025 onward, both developers used AI coding assistants under both conditions. The AI tools and team composition are the controlled variables. The programming language is not: the structured condition used Python/FastAPI following a specified architecture (Honest Code framework, HTMX for frontend); the unstructured condition used React (TypeScript/JavaScript). This divergence is itself a data point: without architectural specification, the AI assistants defaulted to React, the framework most represented in their training data.

3.2 Application A: Platform Description

Application A is a production enterprise platform for Canadian wealth management (NI 31-103, CFR regulatory compliance). The following table summarizes its codebase by language and component.

Language	IDD	IAM	Data Pipeline
Python (production)	108,333	47,298	16,193
Python (tests)	8,464	62,164	315
HTML/Jinja2 templates	45,344	–	–
JavaScript	9,569	–	–
CSS	15,087	–	–
YAML/Docker/Config	3,299	–	234
BDD Feature files	3,859	–	506
Total	**194,955**	**109,462**	**17,248**

The platform comprises three codebases: IDD (the user-facing application), IAM (identity and access management service), and a financial data pipeline that acquires, normalizes, and serves 27,000+ financial data records from multiple sources. The system runs as a 5-service Docker orchestration with separate configurations for development, testing, and production, backed by PostgreSQL, Redis (caching and rate limiting), and per-request cryptographic authentication via a zero-trust API where every data request is signed and verified. The data pipeline evolved through six phases over 13 months across four repositories (448 commits). All line counts exclude JSON data files, virtual environments, and generated assets.

3.3 The Independent Variable: Process Authority

The independent variable is not "structured process vs. no process" in the abstract. It is who controlled the development process.

Condition 1 (structured SDLC): The development process was designed and controlled by the technical architect (Developer A). The process comprised:

- Written functional specifications before implementation
- Behavioral test definitions (BDD) before code

- Test-first implementation
- Architectural review of AI-generated output in real time
- Selective sampling and targeted questioning of generated code
- Pre-commit hooks enforcing formatting, linting, template validation, and automated test execution
- 13 automated quality pipelines

Condition 2 (business-directed process): Control of the SDLC shifted from the technical architect to non-technical stakeholders. Business priorities (demo readiness, feature delivery timelines, stakeholder presentations) determined what was built and how. The structured SDLC constraints were not applied. The same developers used the same AI tools without the specification, testing, review, and quality-gate requirements.

This shift was not a decision by the developers to skip process. It was an organizational decision imposed by stakeholders prioritizing business deliverables over technical process. This distinction matters for experimental design: it eliminates self-selection bias. The developers did not choose the unstructured condition. It was imposed on them.

3.4 The Dependent Variables

Enterprise software quality was measured across 18 dimensions based on generally accepted software industry categorizations of enterprise application maturity. Each dimension maps to published audit and compliance frameworks: SOC 2 Trust Services Criteria, NIST SP 800-53, OWASP ASVS, DORA metrics, CIS Benchmarks, and CNCF best practices. Each dimension was assessed against the minimum threshold a financial services auditor or technology risk committee would apply, as stated in those frameworks.

The 18 dimensions span security architecture (entitlements, authentication, inter-service security), data architecture (multi-tenancy), compliance engineering (audit infrastructure), operational security (rate limiting, configuration and secrets management), performance engineering (caching), operations (notifications), DevOps (CI/CD), infrastructure (containerization), software architecture (dependency injection, pattern sophistication, architectural philosophy), governance (live documentation), process engineering (SDLC with AI safeguards), lifecycle management (tech debt management), and software development (UX/accessibility).

These dimensions were validated by reverse-mapping against three frameworks financial services organizations use during vendor assessments: the Shared Assessments SIG Questionnaire (21 risk domains, 1,936 questions), the FFIEC IT Examination Handbook (11 booklets), and OSFI B-13 Technology and Cyber Risk Management (3 domains, 17 principles). Every application-level examination area in all three frameworks maps to at least one of the 18 dimensions.

Productivity was measured in function points per FTE-month using three independent estimation methods plus a verified lower bound: 1. Lines of code backfiring using Capers Jones language-specific ratios (+/- 30% variance) 2. Feature/screen counting using IFPUG heuristics 3. Commit cadence as a consistency proxy 4. BDD scenario inventory as a verified floor (no estimation required)

3.5 The Two Natural Experiments

Experiment 1: Application A vs. Application B. Both applications target the same domain (Canadian financial services). Application A was built under Condition 1 (structured SDLC) in Python/FastAPI. Application B was built under Condition 2 (business-directed process) in React. The team and tools were constant; the technology stack diverged as a consequence of the process change.

Experiment 2: Codebase C before and after AI-specific SDLC introduction. Developer B was the sole contributor to a third codebase (an identity and access management service). The early commit history (February–March 2025) shows structured manual development with planning documentation visible in the commit record. AI coding assistants were introduced in approximately March 2025 without an AI-specific SDLC; the developer used AI tools within the existing manual development process. The result was AI-generated code that passed the developer's existing quality checks but accumulated structural debt invisible to manual review: test files of 1,000–3,000 lines each for single database models, duplicated controllers, and abandoned data artifacts. When Developer A introduced the AI-specific SDLC (specification constraints on AI output, architectural review of generated code, BDD-first development, pre-commit quality gates), the quality transition was immediate and visible in the commit record: 175,881 lines of AI-generated bloat were eliminated in a single month, replaced by properly structured code with organized test suites, cryptographic authentication, and containerized deployment. The variable that changed was not "structured vs. unstructured" development. It was "structured manual process vs. structured AI-specific process": the addition of constraints designed specifically for AI-generated code.

4 Results

4.1 Enterprise Dimension Comparison

Application A (structured SDLC) satisfies all 18 dimensions. Application B (business-directed process) satisfies 2 of 18.

Application B is a React/TypeScript application with an Express.js backend and SQLite databases. Its git-verified metrics: 326 commits over 10 weeks (January 7 to March 18, 2026), 1,026,484 lines added, 470,501 lines deleted (45.8% deletion ratio), contributed by Developer A (217 commits, 66.6%), Developer B (100 commits, 30.7%), and an AI code generation service (13 commits, 4.0%). The application contains 12 user-facing screens, 65 API endpoints across 11 route files, and approximately 38,000 lines of production code.

Application B is functional software. It is deployed as a live demo with working prospect accounts. The 428 function points represent real, usable features: equity screeners, fund detail pages, due diligence report generation, admin dashboards, file management. Prospects interact with it. The business uses it to demonstrate product capability. In this narrow sense, the unstructured condition produced working software.

The distinction is that working software and sellable software are different things in regulated industries. Application B has no automated tests, no CI/CD pipelines, no code review process (all code pushed directly to default branch), no structured audit logging, no rate limiting, no container orchestration, no pre-commit quality gates, and no behavioral specifications. It stores passwords in plain text. It has partial authentication (HMAC signature verification) and basic configuration management via environment variables, satisfying 2 of 18 dimensions partially.

The 16 missing dimensions are not marginal shortfalls. They are absences. Plain text password storage alone disqualifies a codebase from any financial services technology assessment; the remaining 15 absences compound the finding. More critically, these absences cannot be remediated by retrofitting controls onto the existing codebase. Authentication, entitlements, audit infrastructure, and multi-tenancy must be architectural decisions, not afterthoughts. Application B's 428 functional FP must be discarded and rebuilt on a compliant foundation, which is what the structured SDLC produced as Application A. The business value of Application B is limited to its current role as a sales demonstration tool. Its enterprise-ready function point count is zero.

Application B was not produced in isolation. It was the intended output of a broader R&D pipeline that included an AI-powered design tool (Figma Make, which generated the initial React code from business user prompts) and three successive attempts to build a converter tool that would transform the React output into the specified FastAPI/HTMX architecture. The converter was rewritten three times in 37 calendar days (December 13, 2025 to January 19, 2026) across three separate repositories (37, 10, and 7 commits respectively), with each iteration abandoned and restarted. Six additional unversioned filesystem copies of intermediate states were found with no git history. The converter project was ultimately abandoned, and the structured SDLC team built Application A directly in FastAPI/HTMX from scratch.

This R&D pipeline consumed 1.5 FTE (full-time CTO plus half-time CEO) over 3.2 elapsed months (4.8 FTE-months). Its total git-tracked output across all repositories: 380 commits, 1,153,069 lines added, 483,991 lines deleted. Zero test files. Zero CI/CD pipelines. Four repositories abandoned.

4.2 Productivity Measurement

Four independent estimation methods were applied to Application A's codebase, each with different assumptions and variance. Where the estimates converge, confidence is substantially higher than any single method would provide. All figures exclude JSON data files, virtual environments, and generated assets.

4.2.1 Method 1: LOC Backfiring by Language

Function point "backfiring" estimates function points from an existing codebase by dividing lines of code by language-specific conversion ratios. Capers Jones, the leading researcher in this field, provides these ratios. Jones himself cautions that this method has +/- 30% variance, but it is the best published method for estimating function points from code already written.

Language	LOC/FP (Jones)	IDD LOC	IAM LOC	Est. FP
Python (production)	53	108,333	47,298	2,936
HTML/Jinja2	~34 (markup)	45,344	–	1,334

Language	LOC/FP (Jones)	IDD LOC	IAM LOC	Est. FP
JavaScript	47	9,569	–	204
BDD/Gherkin	~40 (est.)	3,859	–	96
Pipeline Python	53	–	–	306
Total				**4,876**

At +/- 30% variance: upper bound 6,338 FP, lower bound 3,413 FP. Using the conservative lower bound: 3,413 FP / 17.5 FTE-months = 195 FP/FTE-month. Using the midpoint: 4,876 / 17.5 = 279 FP/FTE-month.

4.2.2 Method 2: Feature/Screen Counting

A second approach measures delivered features rather than code volume: pages a user can visit, API endpoints a client can call, data models the system manages. This method is more conservative because it counts what the user sees, not what the developer wrote, and it undercounts infrastructure work (authentication, rate limiting, caching, CI/CD, Docker orchestration) that produces no visible screens but consumes significant engineering effort.

Asset	IDD	IAM
User-facing pages	26	0 (API only)
API route modules	23	10
Service modules	154	39
Controllers	9	15
Data models	–	101
CI/CD pipelines	13	0
BDD scenarios	31	–
Docker services	5	–

Using IFPUG counting heuristics (screens ~5–15 FP, API endpoints ~3–8 FP, data models ~7–10 FP): IDD ~2,140 FP, IAM ~1,200 FP, combined ~3,340 FP. At 17.5 FTE-months: 191 FP/FTE-month.

4.2.3 Method 3: Commit Cadence as Consistency Proxy

The first two approaches measure total output. This third approach measures consistency: a team that delivers 5,000 function points in a single sprint and then stalls for six months has a different profile than a team that delivers steadily over 13 months. Commits are the weakest productivity proxy (a commit can be one line or 5,000 lines), but the most useful for demonstrating sustained cadence. The team sustained 10+ commits per FTE-day across 368 FTE-days. This is not a burst; it is a sustained cadence over 13 months.

4.2.4 Method 4: BDD Verified Floor

The first three approaches involve estimation. Application A's 31 BDD feature files contain 468 scenarios, each declaring a specific system behavior in plain English and verified by an automated test. These scenarios cover both user-facing functionality and system-level requirements: query optimization (35 scenarios), translation engines (49 scenarios), authentication components (34 scenarios), caching (12 scenarios), data processing controllers (21 scenarios), and component systems (65 scenarios), alongside UI specifications for tables, screeners, and charts. At IFPUG heuristics (4–6 FP per scenario): 1,872–2,808 FP. This is a verified floor requiring no estimation. At peak coverage (before a v2 architectural restructure), the codebase contained 119 feature files with 1,887 scenarios. See Section 4.5 for the full validation argument.

4.2.5 Cross-Method Summary

Method	Est. FP	FTE-months	FP/FTE-month
LOC backfiring (midpoint)	4,876	17.5	279
LOC backfiring (lower bound, −30%)	3,413	17.5	195
Feature/screen counting	3,340	17.5	191

Method	Est. FP	FTE-months	FP/FTE-month
BDD verified floor (468 scenarios)	1,872–2,808	17.5	107–160 (floor)

Industry benchmarks for comparison: Industry median 7–10 FP/dev/month (Capers Jones/SPR, 26,000+ projects; QSM/SLIM, 13,000+ projects; ISBSG, 10,000+ projects). Published elite agile benchmark: 20–26 FP/dev/month.

The two LOC-independent methods (feature/screen counting at 191 FP/FTE-month and BDD floor at 7–11 FP/FTE-month for a fraction of the system) confirm that the LOC backfiring estimate is not an artifact of AI verbosity. The conservative estimate of 227 FP/FTE-month used throughout this paper (3,977 FP, the geometric mean of the LOC lower bound and feature counting methods) places the team at 9–11x the published elite benchmark for manual development.

4.3 Application B Productivity

Application B's function points were estimated using screen-based counting (the application is approximately 90% user-facing screens). Using the same IFPUG heuristics applied to Application A: 12 screens at 5–15 FP each (120 FP), 65 API endpoints at approximately 4 FP each (260 FP), and 6 data models at approximately 8 FP each (48 FP). Total: approximately 428 FP.

At 4.8 FTE-months, Application B's productivity was approximately 89 FP/FTE-month. This is 9–12x above the industry median of 7–10 FP/dev/month, confirming that AI coding assistants produce real acceleration under both conditions. The difference between the conditions is not speed. It is what was produced.

Metric	Application A (structured)	Application B (unstructured)
Lines of code generated	1,484,038	1,153,069
Elapsed time	17.5 FTE-months	4.8 FTE-months
Lines generated per month	84,802	360,334
Function points delivered	3,977	428
Enterprise-ready FP	3,977	0
FP/FTE-month	227	89
Financial services audit dimensions (of 18)	18	2
Test files	68	0
CI/CD pipelines	13	0
Repositories abandoned	0	4
Unversioned filesystem copies	0	6

The unstructured condition generated code 4.2x faster by volume (360,334 lines/month vs. 84,802) and delivered nothing that would pass a financial services technology assessment. Speed and enterprise-ready output are inversely correlated when process discipline is absent.

4.4 The 90% Elimination Rate

Across the full dataset, the team produced 2.6 million lines of code and retained approximately 250,000: a 90% elimination rate over 12 months. This is not a cleanup sprint or a one-time refactoring event. It is a sustained, continuous process of generating code with AI assistance, reviewing it against enterprise requirements, and discarding what did not meet the standard.

The industry context makes this figure significant. GitClear's analysis of 211 million changed lines across Google, Microsoft, and Meta repositories (2024–2025) found that cleanup activity collapsed from 25% to under 10% after AI adoption. Code clone rates increased 4x. Churn rose from 5.5% to 7.9%. The industry response to AI-generated code is overwhelmingly to keep it: developers accept the AI's most probable output and move on. SonarSource (2025, 7.9 billion lines) reports organizations accumulate 72 code smells per developer per month, with technical debt compounding rather than decreasing.

This team eliminated 90% of what the AI produced. The 10% that survived constitutes a 248,710-line enterprise platform satisfying all 18 audit dimensions. The elimination was not random; it was the product of real-time architectural review during generation and systematic quality gates after generation. Every retained line represents a human judgment that the code met enterprise standards. Every eliminated line represents a human judgment that it did not.

The 90% figure is also evidence against the concern that AI-assisted productivity metrics are inflated by AI verbosity. If the team were simply accepting verbose AI output, the elimination rate would be low and the codebase would be bloated. The opposite is true: the team generated aggressively, reviewed ruthlessly, and retained only what passed. The resulting codebase is leaner per function point than industry benchmarks for manual development.

4.5 BDD as Independent Validation of Backfiring Estimates

LOC-to-FP backfiring has +/- 30% variance (Capers Jones). A common concern is that AI-assisted code may be systematically more verbose than manual code, which would inflate the LOC-based estimate. However, this concern is inconsistent with the gradient descent argument presented in Section 5.4: if AI output converges toward the mean of the training distribution, then its verbosity converges toward *average* verbosity, not above it. The AI's most probable output is, by definition, the most average output, including in its verbosity per function point. Nevertheless, the +/- 30% variance in backfiring itself warrants independent validation.

Application A's 31 BDD feature files contain 468 scenarios, each a direct, countable specification of a system behavior declared in plain English and verified by an automated test. These are not estimated. They are counted. Critically, the BDD inventory is not limited to user-facing behavior. It covers system-level requirements across the full stack:

Category	Scenarios	Examples
Query optimization & ORM	125	Query optimizer, translation cache, YAML integration
Translation engine	49	Internationalization, locale handling
Authentication & forms	34	Password input, auth components
Component systems	65	Atoms, data, navigation, JS, UI components
Table system	123	Accessibility, columns, formatting, pagination, sorting, state
Screener & integration	38	Unified screener, YAML screener, FastAPI charts
News & data services	21	News controller, DTOs
Caching & storage	13	Chart cache models, avatar storage
Total	**468**	

At IFPUG heuristics (4–6 FP per scenario), the BDD inventory yields 1,872–2,808 FP. This floor requires no estimation: every function point is declared in a specification file and verified by an automated test. The floor is conservative because it counts only behaviors with BDD coverage; dimensions such as CI/CD pipeline configuration, Docker orchestration, and rate limiting configuration are real engineering effort that produces no BDD-countable scenarios.

The validation works as follows. The BDD-counted floor of 1,872–2,808 FP, divided by 17.5 FTE-months, yields 107–160 FP/FTE-month. This floor, derived entirely from counted (not estimated) specifications, already places the team at 5–8x the published elite benchmark of 20–26 FP/dev/month, before any backfiring estimate is applied. The backfiring estimate of 3,977 FP falls within the upper range of the BDD floor, confirming that the LOC-based estimate is not inflated by AI verbosity. The two independent methods converge.

At peak coverage before the v2 architectural restructure, the codebase contained 119 feature files with 1,887 scenarios. The reduction from 119 to 31 files during the restructure reflects architectural consolidation, not loss of coverage: the v2 rebuild reorganized the codebase and re-specified behaviors under the new architecture.

4.6 Cost Analysis

To avoid disclosing confidential compensation, this analysis uses published industry benchmarks for both the actual team and the hypothetical comparison team. Glassdoor (2025) reports the US median for senior full-stack Python developers at US$144K/year base salary. At 1.4x fully-loaded multiplier (MIT Sloan/Hadzima: payroll taxes, benefits, equipment; conservative, excludes recruiting and management overhead), this yields US$202K/year or US$16.8K/month per developer.

The Buckler team comprised 2.0 FTE over 17.5 FTE-months (~8.75 elapsed months per developer on average). At the Glassdoor median fully-loaded rate: 2.0 developers x US$16.8K/month x 8.75 months = approximately US$294K.

An industry-median team producing the same output (3,977 FP at the conservative estimate) at 8.5 FP/dev/month would require 27 developers over 17.5 months. At the same Glassdoor fully-loaded rate: 27 x US$16.8K x 17.5 = approximately US$7.9M.

Using the same published benchmark for both teams, the structured AI-assisted team produced at approximately 4% of the cost of an industry-median team while satisfying all 18 enterprise dimensions. The following table places this result in the full spectrum of industry productivity tiers. All costs use the same Glassdoor fully-loaded rate (US$16.8K/month) and the conservative function point estimate (3,977 FP).

Tier	FP/dev/ month	Team Size	Cost (USD)	Expected Process Profile
Bottom quartile	3.5	65	$19.1M	Ad hoc (CMMI L1). No formal test plans, no version control discipline. 75% of global orgs operate here (Jones).
Industry median	8.5	27	$7.9M	Basic project mgmt (CMMI L1–2). Requirements documented but not formally managed. US average.
Upper quartile	15	15	$4.4M	Org-wide standard process (CMMI L3). Formal peer reviews, defined engineering processes.
Elite agile	23	10	$2.9M	Data-driven quality mgmt (CMMI L3–4). Quantitative objectives, predictive models. Best-in-class manual dev.
Structured AI	**227**	**2.0**	**~$294K**	Structured AI-assisted SDLC (CMMI L2–5 practices). BDD-first, 13 quality pipelines, 100% coverage, 55% cleanup ratio, real-time architectural review.

The tier comparison reveals two findings. First, the cost differential is not marginal: the structured AI-assisted team produced at ~4% of the industry-median cost and ~10% of the elite agile cost. Second, the CMMI maturity levels predict the quality profile at each tier. An industry-median team at CMMI L1–2 would deliver approximately 0.75–1.05 bugs per function point (Jones), producing roughly 3,000–4,200 bugs across the same 3,977 FP with no guarantee of enterprise compliance. The structured AI-assisted team operates at L4–5 practices (predicted: $<$0.32 bugs/FP, $>$93% defect removal before release).

4.7 Planning Before Implementation

The structured SDLC's most distinctive feature is not its quality gates but the time invested in planning before any code is written. The git history makes this measurable: commits touching only documentation files (.md, .feature) can be distinguished from commits touching code files (.py, .html, .js, .css).

Of 194 active days in the IDD repository, 43 (22.2%) were planning-oriented (documentation-only or documentation-heavy commits). Ten distinct planning clusters were identified (consecutive days where documentation commits dominated), each followed by an implementation burst within 1 day. The largest cluster (7 days, February 6–16, 2026, 68 documentation commits) preceded the v2 rebuild's most sustained implementation period: 38 code commits on the first day alone, followed by 84 commits over 6 days implementing authentication middleware, controllers, repositories, and services.

The February 2026 v2 rebuild phase illustrates the pattern at its clearest: 52% of February commits were documentation (architecture specifications for 18 planning domains). March shifted to 84% code, with 148 implementation commits across 7 working days. The planning investment was not idle time. It was the specification work that constrained the AI's output and enabled the implementation velocity that followed.

This pattern has no counterpart in Application B's commit history or in the unstructured R&D pipeline. Those repositories show code commits from the first day forward, with no planning clusters, no specification phases, and no documentation-heavy periods preceding implementation.

4.8 Activity Ratios: Documentation, Coding, and Refactoring

If process discipline is the key variable, then how that discipline allocates effort across documentation, coding, and refactoring is a concrete, measurable characterization of what "structured SDLC" means in practice. No published study has reported these ratios for AI-assisted enterprise development. The git history provides them directly: commits

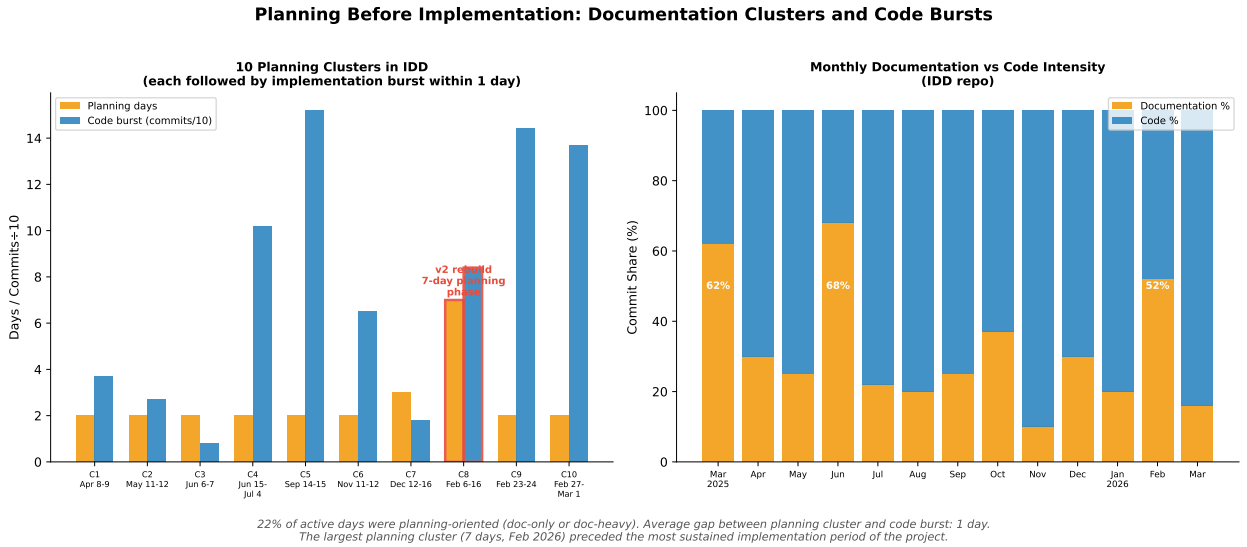


Figure 6: Left: 10 planning clusters identified in the IDD commit history, each followed by an implementation burst within 1 day. Right: monthly documentation vs. code commit intensity. Planning-heavy months (Mar 2025, Jun 2025, Feb 2026) each preceded major implementation phases.

were classified by the file types they touch (.md, .feature, .txt for documentation; .py, .html, .js, .css, .yaml, Dockerfile for code) and by their deletion ratio (lines deleted as a percentage of lines added).

Metric	Structured (A)	Unstructured (B)
Total commits	2,672	385
Doc-only commits	358 (13.4%)	1 (0.3%)
Code-only commits	1,575 (58.9%)	326 (84.7%)
Mixed (doc + code)	482 (18.0%)	23 (6.0%)
Doc lines as % of all lines added	41.6%	3.1%
Code delete/add ratio	76.9%	44.6%
Net-negative commits	385 (18.7%)	58 (16.6%)
High delete ratio (>40%)	725 (35.2%)	132 (37.8%)

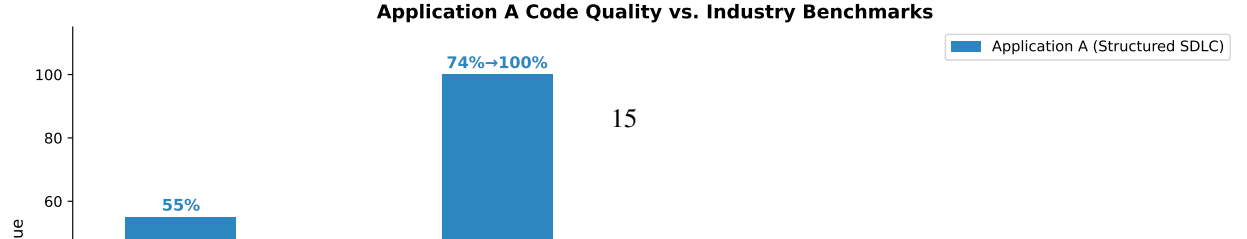
Three findings emerge. First, the structured condition dedicates 41.6% of all lines added to documentation (specifications, BDD features, architecture documents), versus 3.1% in the unstructured condition. By commit count, 13.4% of structured commits are documentation-only versus 0.3% unstructured. This is the measurable cost of "specifications before code."

Second, the structured condition's code delete/add ratio is 76.9% (77 lines deleted for every 100 added), versus 44.6% for the unstructured condition. The IDD repository alone has an 89.0% ratio. This confirms that the structured SDLC produces sustained refactoring, not one-time cleanup, and provides project-level context for the GitClear finding that industry-wide cleanup activity collapsed to under 10% after AI adoption.

Third, mixed commits (documentation and code in the same commit) are 3x more prevalent in the structured condition (18.0% vs. 6.0%), indicating that documentation is maintained alongside code changes rather than deferred or omitted.

The unstructured condition's profile (84.7% code-only commits, 0.3% doc-only, 3.1% documentation lines) is consistent with the "code first, document never" pattern that GitClear, DORA, and He et al. have documented at industry scale. The structured condition inverts this distribution.

4.9 Code Quality Signals



4.10 Maturity Assessment

To characterize the maturity of the structured SDLC independently of the 18-dimension outcomes, the team's documented practices were mapped against CMMI (Capability Maturity Model Integration) practice areas. CMMI is a 5-level scale assessing organizational process maturity; Level 1 is ad hoc, Level 5 is continuously optimized. CMMI measures organizational breadth (how many teams follow the process), which is not directly applicable to a two-person team. What can be assessed is which level's *practices* the team's process satisfies.

CMMI Practice Area	Evidence	Level
Requirements Mgmt	Written functional specifications before implementation	L2
Configuration Mgmt	Git with 13 CI/CD pipelines, pre-commit hooks, Docker	L3
Verification	468 BDD scenarios across 31 feature files, 68 test files, 100% coverage, smart test runner	L3
Process & Product QA	Pre-commit formatting/linting, template validation, architectural review of AI output	L3
Measurement & Analysis	FP tracking, commit metrics, cleanup ratios, performance benchmarks	L4
Quantitative Project Mgmt	Data-driven sprint planning, velocity measurement across repos	L4
Causal Analysis & Resolution	Active tech debt elimination (175K+ lines), AI-generated bloat detection and removal	L5

The practices span Level 2 through Level 5. Capers Jones's data predicts that a team operating at Level 4–5 practices should catch over 93% of bugs before release and deliver fewer than 0.32 bugs per function point. Application A's 100% test coverage, 13 automated quality pipelines, and 55% cleanup ratio (indicating systematic quality improvement) are consistent with that prediction.

4.11 The IAM Transition (Experiment 2)

Developer B's commit history on the identity and access management codebase captures a three-phase transition:

Phase 1 (February–March 2025): Human-written code establishing the quality ceiling. Developer B wrote 100% human-generated code: database models, migrations, ERD generation, project structure, pull request workflow with named branches. This code established the architectural patterns, naming conventions, and quality standards that all subsequent code would be measured against. In generation loss terms, this phase set the quality ceiling for the codebase: the context that AI assistants would absorb when introduced. Critically, this context was available only within the shared Docker image containing IAM and IDD (formerly named flanker): AI assistants operating in these codebases absorbed the established patterns. The unstructured repositories (react2htmx, IddAdvisor) were separate codebases with no access to this context. The AI operating in those repos had no quality ceiling to absorb, which is why their output converged on the training distribution mean rather than on the established enterprise patterns.

Phase 2 (March–May 2025): AI-assisted development without an AI-specific SDLC. AI coding assistants were introduced, but within the existing manual development process. The developer applied the same quality standards used for manual coding. The result: AI-generated test files of 1,000–3,000 lines each for single database models: code that appeared to work and passed the developer's existing review but accumulated structural debt invisible to manual review techniques. This matches GitClear's industry-scale finding: AI output requires different review methods than manual code because the failure modes are different (volume without design rather than bugs from inattention).

Phase 3 (May–June 2025): AI-specific SDLC introduced. Developer A introduced constraints designed specifically for AI-generated code: specification constraints on AI output, architectural review of generated code, BDD-first development, pre-commit quality gates. The immediate consequence was a systematic cleanup of the Phase 2 output.

After structured SDLC introduction: In June 2025, Developer B eliminated 175,881 lines in 9 commits (Figure 8). The monthly line-level data for the IAM codebase captures the transition:

Month	Commits	Lines Added	Lines Deleted	Net
2025-02	19	36,087	13,841	+22,246
2025-03	7	5,350	3,648	+1,702
2025-05	6	24,805	24,481	+324
2025-06	**131**	**505,283**	**263,643**	**+241,640**
2025-07	11	11,202	420	+10,782
2025-08	26	50,819	3,054	+47,765

The June 2025 spike (131 commits, 505K lines added, 263K deleted) represents the full rewrite/restructuring cycle after SDLC introduction. The 9 largest deletion commits itemize what was eliminated:

Commit	Deletions	What Was Eliminated
b825f35	−38,042	40 bloated AI-generated test files (500–2,000+ lines each)
20a4dfe	−33,539	Old controller/route/DTO code replaced by working endpoints
90ca5b8	−19,165	9 massive test files during SQLAlchemy 2.0 conversion
7272d9e	−19,160	Formatting/linting cleanup pass
8f60e64	−15,468	26 unnecessary CSV seed files
bd6dd21	−14,113	6 bloated test files replaced by properly organized suites
718c122	−12,797	Old test files replaced after model changes
575ab9c	−12,008	10 old test files replaced (achieved 64% coverage milestone)
e939152	−11,588	Old models replaced by SQLAlchemy 2.0 typed models
Total	**−175,881**	

The deleted code was replaced by properly structured SQLAlchemy 2.0 models, organized test suites (separated by type), cryptographic authentication between services, audit trails, and containerized deployment. Net +241K lines in June confirms this was replacement at industrial scale, not deletion alone. SonarSource (2025) reports that the average developer introduces 72 code quality problems per month; Developer B eliminated 175K+ lines of accumulated problems in a single month under the structured SDLC.

The quality shift was caused by the introduction of the structured SDLC, not by changing the developer, the tools, or the language.

5 Analysis

5.1 Process as the Key Variable

The two natural experiments converge on the same finding. In Experiment 1, the same team using the same tools produced radically different enterprise-quality outcomes under different process conditions. In Experiment 2, the same developer on the same codebase produced radically different code quality before and after structured process introduction. The variable that changed in both cases was the development process.

This is consistent with DORA's (2025) characterization of AI as an "amplifier," but provides a mechanism that survey data cannot: the structured SDLC constrains the AI's output through specifications, tests, architectural review, and automated quality gates. Without these constraints, the AI generates volume (lines, files, apparent features) without the enterprise characteristics (security controls, audit trails, test coverage, deployment automation) that regulated buyers require.

5.2 The Iceberg: Visible Output vs. Auditable Infrastructure

The 18 enterprise dimensions can be grouped by lifecycle category to reveal a distribution with implications for how AI-assisted productivity is measured.

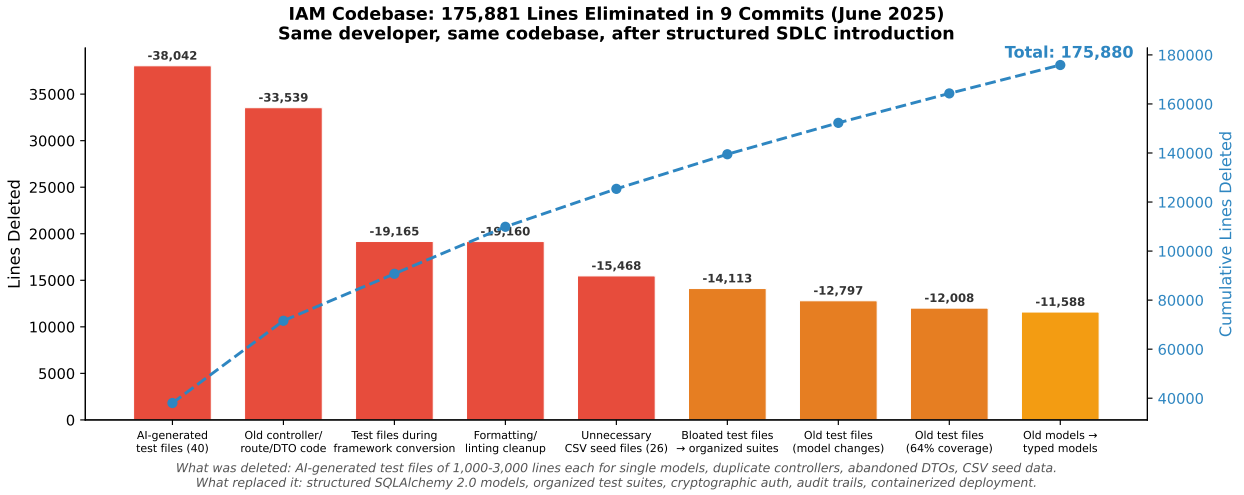


Figure 8: IAM codebase tech debt elimination: 175,881 lines removed in 9 commits during June 2025. The deleted code consisted of AI-generated test files of 1,000–3,000 lines each, duplicate controllers, and abandoned data. The replacement code: structured SQLAlchemy 2.0 models, organized test suites, cryptographic authentication, audit trails, containerized deployment. Same developer, same codebase, after structured SDLC introduction.

Lifecycle Category	Dimensions	User-Visible?
Security Architecture	Entitlements, Authentication, Inter-Service Security	No
Data Architecture	Multi-Tenancy	No
Compliance Engineering	Audit Infrastructure	No
Operational Security	Rate Limiting, Config & Secrets	No
Performance Engineering	Caching	No
Operations	Notifications	Partially
DevOps / Release Eng.	CI/CD	No
Infrastructure	Containerization	No
Software Architecture	Dep. Injection, Pattern Soph., Arch. Philosophy	No
Governance	Live Documentation	No
Process Engineering	SDLC with AI Safeguards	No
Lifecycle Management	Tech Debt Management	No
Software Development	**UX from Code**	**Yes**

Of 18 dimensions, 1 is visible to end users. The remaining 17 are infrastructure that a financial services auditor evaluates before the user-facing code is examined. This distribution explains why Application B (which is functional, deployed, and used for live demonstrations) scores 2 of 18: it has the visible layer (screens, features, interactions) but none of the auditable infrastructure beneath it.

This distribution also explains why productivity metrics focused on visible output (screens delivered, features completed, lines of code generated) systematically undercount enterprise effort. However, the structured SDLC's use of BDD extends beyond user-facing behavior: the 468 BDD scenarios in Application A cover system-level requirements including query optimization, translation engines, authentication, caching, and component systems (Section 4.5). The BDD verified floor of 1,872–2,808 FP therefore spans multiple dimensions of the iceberg, not just the visible tip. The gap between that floor and the conservative backfiring estimate of 3,977 FP represents the remaining infrastructure dimensions (CI/CD configuration, Docker orchestration, rate limiting, inter-service security) that produce no BDD-countable scenarios. Any team can build screens. The 17 infrastructure dimensions are where applications fail enterprise assessments, and they are the dimensions that AI-without-process does not produce at all.

5.3 The Three Requirements

The data suggests three necessary conditions for productive AI-assisted enterprise development, any one of which, if absent, produces failure:

1. **Structured process discipline:** Specifications before code, tests before implementation, architectural review during generation, automated quality gates before commit.

2. **Elite-level architectural expertise to enforce it:** The structured SDLC requires a specific set of skills that are observable and measurable, though not commonly discussed in the AI coding literature. During generation, the architect monitors AI output in real time, watching for architectural drift, the gradual divergence from the specification that compounds into structural damage if not caught within the first few lines. This requires recognizing good architecture on sight, which is a function of experience building, shipping, and maintaining enterprise systems, not simply experience writing code. After generation, the architect reviews through targeted questioning ("why did you use inheritance here instead of composition?") and selective sampling (reading specific modules chosen for their architectural significance, not random spot checks). These review techniques are themselves a skill: knowing which questions to ask and which code to sample requires understanding what a financial services auditor, a security reviewer, or a future maintainer will evaluate. The METR study's finding that experienced open-source developers were 19% slower with AI is consistent: coding experience is necessary but not sufficient. The additional requirement is architectural judgment: the ability to evaluate whether generated code satisfies systemic constraints (security boundaries, coupling profiles, compliance requirements) that are invisible at the function level.

3. **A clean starting codebase that sets the context for AI generation:** AI coding assistants do not generate code in a vacuum. They operate within the context of the existing codebase, absorbing its naming conventions, architectural patterns, module boundaries, and structural constraints. An AI operating in a well-designed codebase absorbs those conventions and produces code that follows them. An AI operating in a poorly structured codebase absorbs those patterns and reproduces them. The starting point is the ceiling.

   This is not speculation. Shumailov et al. (2024, *Nature*) demonstrated that generative models trained on their own output undergo irreversible "model collapse": the tails of the original distribution disappear with each generation. Applied to code: each iteration of AI-generated code loses architectural subtlety, edge case handling, and structural discipline. Shukla et al. (2025, IEEE-ISTAS) measured the rate: 37.6% increase in critical vulnerabilities after just 5 iterations. The degradation is cumulative and directional.

   Application A's codebase was designed with all 18 enterprise dimensions from the beginning, not retrofitted. Authentication, entitlements, audit infrastructure, rate limiting, and the remaining dimensions were architectural decisions made before the first line of AI-generated code was written. This means the AI assistant operated within a context that enforced enterprise constraints: when generating a new endpoint, it absorbed the existing pattern of entitlement decorators, audit logging calls, rate limit configurations, and HMAC-signed inter-service calls. The generated code followed these patterns not because the AI understood their purpose, but because the codebase context made them the path of least resistance.

   Application B's codebase had none of these patterns. The AI assistant operating in that context absorbed the absence of entitlements, audit trails, and quality gates, and reproduced it. The tool was the same. The context was different. The generation loss research explains why this gap widens over time rather than narrowing: each generation of AI output in a degraded codebase moves further from enterprise quality, while each generation in a well-designed codebase reinforces the existing patterns.

5.4 Fighting the Gradient

The three requirements are not merely additive safeguards. They are a continuous correction force against a mathematical tendency. As noted in the abstract, gradient descent converges toward the mean of the training distribution. For code generation, this means the AI's weights and biases perpetually pull its output toward the most average code in its training set: code that, by definition, does not satisfy enterprise requirements. Left uncorrected, every generated function, every suggested pattern, every completed module drifts toward the median.

Producing elite results with AI assistance therefore means fighting the AI at every step. Every specification constrains it away from the mean. Every real-time architectural intervention catches it drifting back. Every pre-commit quality gate rejects average output. Every review question ("why did you use inheritance here instead of composition?") forces it to justify a deviation from its most probable output. The structured SDLC is not a passive quality framework. It is an active, continuous correction mechanism that counteracts the tool's statistical gravity.

This is why the expertise requirement is so high. The correction must happen in real time, before mean-reversion compounds across modules. A reviewer who checks code after the fact finds individual defects. An architect who monitors generation in real time prevents the systemic drift that turns enterprise code into average code, one statistically probable decision at a time. The METR finding (that experienced developers were 19% slower with AI) is consistent: the developers in that study were experienced with code but were not counteracting the gradient. They accepted the AI's most probable output and spent more time debugging the consequences than they would have spent writing the code manually.

The Buckler data demonstrates what happens when this correction force is removed while team and tools remain constant: the enterprise quality outcomes collapse from 18/18 to 2/18 dimensions satisfied. The AI did not become worse. The correction stopped.

5.5 The Exogenous Process Change

The process shift from Condition 1 to Condition 2 was not a developer decision. It was an organizational decision: non-technical stakeholders assumed control of the SDLC, directing the team toward business deliverables (demos, feature presentations, stakeholder requests) without the structured SDLC constraints. The developers did not choose to abandon specifications, testing, and code review. Those activities were displaced by business-directed urgency.

This distinction is methodologically significant. In most observational comparisons of development practices, self-selection bias confounds the results: teams that adopt structured processes may be more disciplined to begin with. In this case, the same team operated under both conditions, and the transition was imposed externally. The process variable was exogenous to the developers' preferences.

6 Threats to Validity

6.1 Internal Validity

Author as subject. The author designed the structured SDLC, served as architectural reviewer, and wrote this analysis. This is a conflict of interest. The mitigation is that all metrics reported in this paper are derived from git commit history and source code analysis, not from subjective assessment. Commit counts, line additions, line deletions, file counts, and commit timestamps are objective and independently verifiable. The 18-dimension assessment uses published frameworks (SOC 2, NIST 800-53, OWASP ASVS) with stated minimum thresholds.

No randomization. This is a natural experiment, not a randomized controlled trial. The team was not randomly assigned to conditions. The mitigation is that the process change was exogenous (imposed by organizational authority, not self-selected by the developers), which reduces but does not eliminate selection bias.

Confounding variables. The two applications differ in scope and purpose, not only in process. Application A is a larger, more complex platform. It is possible that the complexity itself forced more structured development. The IAM transition (Experiment 2) partially addresses this concern: the same codebase, same scope, different process, different outcomes.

6.2 External Validity

Team size. A two-person team may not generalize to larger organizations. The structured SDLC described here relies on a single architectural reviewer with direct oversight of all generated code. This may not scale without modification.

Domain. The findings are from financial services software built for regulatory compliance. The 18 dimensions are not domain-specific; they represent standard dimensions of enterprise software readiness (authentication, testing, CI/CD, containerization, audit trails, etc.) that apply to any enterprise application. The financial services audit mapping (SOC 2, NIST 800-53, OWASP ASVS, OSFI B-13, FFIEC, SIG) validates completeness against the most demanding regulatory regime, but the dimensions themselves would appear on any enterprise technology assessment. The underlying thesis (process discipline determines AI-assisted development quality) and the measurement framework should both generalize to enterprise software outside financial services.

Technology stack divergence. The structured condition used Python/FastAPI; the unstructured condition used React. This is a confounding variable: the quality differences could be partially attributed to the technology choice rather than the process. The mitigation is twofold. First, the technology divergence is itself a consequence of the absent process: without an architectural specification, the AI assistants defaulted to React, which is a finding about unstructured AI-assisted development, not an independent variable. Second, Experiment 2 (the IAM transition) controls for language –

the same developer, same codebase, same language (Python), with only the process changing – and produces the same quality transition.

6.3 Construct Validity

Enterprise dimensions defined by the author. The 18 dimensions are based on generally accepted software industry categorizations and mapped by the author to published frameworks. The mitigation is the reverse-mapping validation: every application-level examination area in the SIG Questionnaire, FFIEC IT Examination Handbook, and OSFI B-13 maps to at least one dimension. The dimensions were not selected to favor the structured application.

Function point estimation. The LOC-to-FP backfiring method has +/- 30% variance (Capers Jones). As argued in Section 5.4, gradient descent's convergence toward the training distribution mean implies that AI-generated code is not systematically more verbose than average manual code; it *is* average, by mathematical construction. The +/- 30% variance remains a concern independent of AI verbosity. The feature/screen counting method (191 FP/FTE-month) is independent of LOC and produces a consistent result. The BDD verified floor (107–160 FP/FTE-month from 468 counted scenarios covering both user-facing and system-level requirements) independently confirms the backfiring estimate without requiring any estimation.

6.4 Proprietary Codebase

The source code is proprietary and cannot be published. The author invites independent audit of the git history and source code under appropriate confidentiality arrangements. To specifically facilitate verification of the function point estimates, which form the basis for the reported productivity figures, the author extends a particular invitation to any IFPUG Certified Function Point Specialist (CFPS), or equivalent accredited professional with demonstrated expertise in IFPUG function point counting, to review the codebase under a standard mutual non-disclosure agreement. Interested parties should contact the corresponding author (`adam.wasserman@buckler.ai`) to discuss arrangements. The author commits to publishing any material corrections resulting from such reviews as an erratum or revised preprint.

7 Discussion

7.1 Implications for the AI-Assisted Development Debate

The findings suggest that the contradictions in the AI-assisted development literature are not contradictions at all. They are measurements of different conditions. AI accelerates simple tasks (Peng et al.) and degrades complex system quality (GitClear, DORA, METR) because the studies measured different process conditions. When AI operates within a structured process with expert oversight and a clean codebase, it produces extraordinary output. When it operates without those constraints, it produces volume without quality. The tool is the same. The process is different.

This reframes the policy question from "should organizations adopt AI coding assistants?" to "under what process conditions do AI coding assistants produce enterprise-quality output?" The answer suggested by this data: structured specifications, test-first development, real-time architectural review of AI output, automated quality gates, and a codebase designed with enterprise constraints from the beginning.

7.2 Implications for Organizational Governance

The exogenous nature of the process change in this study has a specific implication: the quality collapse was caused by an organizational decision, not a technical one. When non-technical stakeholders assumed control of the SDLC, the enterprise quality outcomes collapsed even though the team and tools remained constant. This suggests that organizational governance of AI-assisted development is as important as the technical practices themselves. A structured SDLC that can be overridden by business urgency is not a structured SDLC.

7.3 Implications for Platform Economics and Maintenance

Published research on software lifecycle costs has implications for interpreting the structured condition's platform architecture. The literature converges on a consistent finding: building software is the smaller part of the lifecycle cost. Glass (2003, *Facts and Fallacies*, Fact 41) reports 40–80% of lifecycle cost is maintenance. Boehm (1981) estimated 60–70%. Capers Jones (2007) found 72% is post-delivery work. Enterprise TCO studies put it at 75–85% ongoing vs. 15–25% initial over a 5-year horizon. Stripe (2018, thousands of respondents across 30+ industries) found

developers spend 33% of their time on technical debt. CISQ (2022) attributes US$1.52 trillion in accumulated US technical debt.

Finding	Ratio	Source	Year
Maintenance as % of lifecycle	40–80%	Glass, Fact 41	2003
Maintenance as % of lifecycle	60–70%	Boehm, COCOMO	1981
Post-delivery work	72%	Capers Jones	2007
Enterprise TCO (5-year)	75–85% ongoing	Industry standard	Various
Developer time on tech debt	33%	Stripe	2018
High-debt maintenance premium	+40%	McKinsey	2020
US accumulated tech debt	$1.52T	CISQ	2022

Application A was designed as shared infrastructure: each additional product built for the Canadian wealth management market inherits all 18 enterprise dimensions. The marginal cost of additional products is domain-specific logic only, and every security patch, dependency upgrade, and regulatory update is applied once. On duplicated infrastructure, each fix must be propagated independently across codebases. Ponemon/ServiceNow (2019, 3,000 professionals) found that 60% of breaches involved vulnerabilities where a patch was available but not applied, with an average 12-day delay caused by organizational coordination across systems. The Equifax breach (2017, US$1.4B total cost) resulted from a single patch not applied across systems for 142 days.

Application B's 428 function points must be discarded and rebuilt on a compliant foundation. The development cost of Application B (4.8 FTE-months at Glassdoor median rates: ~US$161K) is therefore 100% wasted from an enterprise lifecycle perspective. The structured condition's higher initial investment in platform architecture is the smaller part of the lifecycle bill; the ongoing cost advantage compounds with each product added to the platform and each security patch applied once rather than propagated.

7.4 Limitations and Future Work

This study provides evidence from a single natural experiment with a two-person team. A complete four-group comparison (AI with and without structured process, manual coding with and without structured process) remains the gold standard for establishing causation. The present data covers the two AI-assisted groups. A complete design would require a manual-coding control, which was not available in this dataset.

The function point estimates, while consistent across three independent methods and validated against a BDD-verified floor, would benefit from independent certification by a certified function point analyst.

Replication across different team sizes, domains, languages, and AI tools would strengthen the generalizability of the findings.

8 Conclusion

No published study has isolated process discipline as a controlled variable in AI-assisted software development outcomes. This paper presents a natural experiment that holds team and tools constant while varying the development process through an exogenous organizational change. The structured condition produced an application satisfying 18 enterprise dimensions covering the criteria financial services technology assessments measure, at 9–11x the published elite productivity benchmark. The unstructured condition generated code 4.2x faster by volume, produced 1.15 million lines across 380 commits in 4.8 FTE-months, abandoned four repositories, and delivered an application satisfying 2 of 18 dimensions with no tests, no quality pipelines, and no code review. Speed and enterprise-ready output were inversely correlated.

The findings are consistent with DORA's characterization of AI as an amplifier but provide granular, project-level evidence that aggregate survey data cannot. The data suggests three multiplicative requirements for productive AI-assisted enterprise development: structured process discipline, elite-level architectural oversight, and a clean starting codebase. When organizational decisions removed the first requirement while the other two remained partially available, the enterprise quality outcomes collapsed.

The contradictions in the AI-assisted development literature are resolved when process discipline is treated as the mediating variable. AI does not make developers faster or slower. It amplifies whatever process discipline – or lack thereof – surrounds the work.

A Enterprise Audit Dimensions

The 18 dimensions used to assess enterprise software readiness in this study are listed below, grouped by lifecycle category. The dimensions are based on generally accepted software industry categorizations of enterprise application maturity. Each dimension maps to published audit and compliance frameworks: SOC 2 Trust Services Criteria, NIST SP 800-53, OWASP ASVS, DORA metrics, CIS Benchmarks, and CNCF best practices. Each was assessed against the minimum threshold a financial services auditor or technology risk committee would apply, as stated in those frameworks. The dimensions were validated by reverse-mapping against the Shared Assessments SIG Questionnaire (21 risk domains, 1,936 questions), the FFIEC IT Examination Handbook (11 booklets), and OSFI B-13 Technology and Cyber Risk Management (3 domains, 17 principles).

#	Dimension	Industry Standard (Minimum)	Application A Evidence	CMMI	B
1	Entitlements	Per-endpoint authorization, deny-by-default, role-based access (Gartner AM MQ)	27 enforcement points, 10 subscription types, security logging on all decisions	L3	Fail
2	Authentication	AAL2 with session binding and secure expiry (NIST SP 800-63B-4)	Per-request multi-factor via zero-trust API. Firebase + JWT + Redis sessions + HMAC-SHA256 request signing	L3–4	Part.
3	Inter-Svc Security	Cryptographic verification, replay protection (CISA Zero Trust v2.0 Advanced)	HMAC-SHA256 signing, timestamp replay protection, per-service key isolation	L3–4	Fail
4	Multi-Tenancy	Tenant isolation at query level, no data leakage (JISEM 2025)	Company context from IAM at middleware layer, query-level scoping, dedicated instance support	L3	Fail
5	Audit Infra.	Tamper-evident, timestamped, queryable records (NIST AU-2/AU-3, SOC 2 CC7.2)	69 audit entry points, SHA256 entry IDs, risk scoring, PII redaction	L3–4	Fail
6	Rate Limiting	Per-endpoint, per-user enforcement with 429 responses (OWASP API4)	Redis sliding window, 5 configurations, Retry-After headers, in-memory fallback	L3	Fail
7	Caching	Tiered TTLs per data volatility, proactive warming (Redis/AWS best practices)	9 TTL configurations, 7 invalidation types, cache warming every 4 min, 12 concurrent fetches	L4	Fail
8	Notifications	Dedicated workers, delivery guarantees, async UI feedback (Gartner EDA)	SSE streaming, Redis queue coordination, modal confirmations, cleanup worker	L3	Fail
9	CI/CD	Automated test, build, deploy, security scan with gating (DORA High/Elite)	13 workflows, pre-commit hooks (formatting, linting, template validation, smart test runner)	L4	Fail
10	Containers	Multi-stage builds, non-root, health checks, env parity (CNCF 2024)	5 Docker services, Alpine multi-stage, non-root, health checks, 3 compose files	L3	Fail
11	Config/ Secrets	Externalized config, secrets never in code, rotatable credentials (CIS, Akeyless)	75 env vars defined, 275 references across 55 files, no secrets in source	L3	Part.
12	Dep. Injection	Explicit registration, singleton lifecycle, protocol contracts (IEEE DI research)	8 singleton services, protocol-based contracts, lazy loading, 50+ services total	L3	Fail
13	Pattern Soph.	High diversity + low grime + problem-pattern alignment (OOPSLA DPD)	Dispatch tables, plugin/hook system, event sourcing, zero-trust API, strangler pattern, config-driven rules	L4–5	Fail
14	Live Docs	Docs updated within 90 days of code changes (DORA 2024)	docs.html system, architecture docs, CLAUDE.md as living guides, 31 BDD feature files (468 scenarios) as executable docs	L3	Fail

#	Dimension	Industry Standard (Minimum)	Application A Evidence	CMMI	B
15	SDLC/AI	Structured review of AI output, quality gates, automated testing (DORA 2024)	Spec-first, BDD-first, test-first, architectural review, audit sampling, pre-commit hooks, 74% to 100% coverage	L4–5	Fail
16	Tech Debt	Active refactoring, TDR under 5%, measurable debt reduction (SonarSource, CodeScene)	175K+ lines eliminated in one month, 55% overall cleanup ratio	L5	Fail
17	Arch. Phil.	Articulated philosophy, consistently applied, conformance enforcement (Brooks, ISO 25010)	16 codified principles, consistent application across all modules	L5	Fail
18	UX from Code	Nielsen heuristic compliance, WCAG 2.2, Core Web Vitals (LCP <2.5s, CLS <0.1)	71 ARIA attributes, accX module (15 types), ~3.3KB JS, nav depth 2, server-rendered 0ms TBT	L3–4	Fail
	Total				**18/2**

Of the 18 dimensions, 1 (UX from Code) is visible to end users. The remaining 17 are infrastructure that a financial services auditor evaluates before the user-facing code is examined. Application B satisfies dimension 2 (authentication) partially (working auth layer but plain text password storage) and dimension 11 (config & secrets) partially (environment variables used but no systematic management). All other dimensions are absent, not partially implemented.

B Monthly Commit Data by Developer

The following tables present monthly commit counts by developer across all structured-condition repositories. Daily commit data (194 unique dates for IDD, 47 for IAM) is available from the authors on request.

Month	Developer A	Developer B	Total
IDD (Application A)			
2025-03	0	8	8
2025-04	0	142	142
2025-05	6	207	213
2025-06	22	0	22
2025-07	129	119	248
2025-08	122	243	365
2025-09	78	170	248
2025-10	69	60	129
2025-11	152	2	154
2025-12	25	111	136
2026-01	0	25	25
2026-02	0	268	268
2026-03	0	173	173
IDD Total	**603**	**1,528**	**2,131**
IAM (Codebase C—Developer B sole contributor)			
2025-02	0	24	24
2025-03	0	10	10
2025-05	0	7	7
2025-06	0	144	144
2025-07	0	14	14
2025-08	0	28	28
IAM Total	**0**	**227**	**227**
Data Pipeline (flanker-demo, sra-data, Buckler-API)			
2025-03 to 06	11	26	45

Month	Developer A	Developer B	Total
2025-10 to 2026-01	89	7	98
2026-02 to 03	41	2	48
Pipeline Total	**141**	**35**	**191**
Grand Total	**744**	**1,790**	**2,549**

Developer B contributed 70.2% of all commits (1,790 of 2,549). Developer A's commit activity is concentrated in periods of architectural work (July–November 2025 on IDD, October–January on the data pipeline), with Developer B sustaining high-volume implementation throughout.

C Dimension-to-Audit-Regime Mapping

The following table maps each of the 18 enterprise dimensions to the specific audit frameworks and regulatory regimes that examine them. This mapping was used to validate completeness: every application-level examination area in all six frameworks maps to at least one dimension.

#	Dimension	Audit Frameworks
1	Entitlements	SOC 2 CC6.1/CC6.3; NIST AC-2/AC-6; OSFI B-13 s.4.3; NI 31-103 s.11.1
2	Authentication	NIST SP 800-63B-4; SOC 2 CC6.1; OSFI B-13 s.4.3; SEC Reg S-P; FINRA Rule 3110
3	Inter-Svc Sec.	CISA Zero Trust v2.0; NIST SC-8/SC-23; OSFI B-13 s.4.4
4	Multi-Tenancy	SOC 2 CC6.1; PIPEDA s.7; OSFI B-13 s.4.3; SEC Reg S-P Rule 248.30
5	Audit Infra.	NIST AU-2/AU-3/AU-10; SOC 2 CC7.2; OSFI B-13 s.4.6; NI 31-103 s.11.5; SEC Rule 17a-4
6	Rate Limiting	OWASP API4; NIST SC-5; OSFI B-13 s.4.4; SOC 2 CC6.6
7	Caching	OSFI E-21 s.3; NIST SC-4; SOC 2 CC7.5
8	Notifications	SOC 2 CC7.3; OSFI E-21 s.3; NIST IR-6
9	CI/CD	SOC 2 CC8.1; NIST SA-11/SA-15; OSFI E-21 s.4; ISO 27001 A.8.25–A.8.33
10	Containers	NIST SP 800-190; CIS Docker/K8s Benchmarks; OSFI B-13 s.4.5; SOC 2 CC6.7
11	Config/Secrets	NIST SC-12/SC-28; SOC 2 CC6.1; OSFI B-13 s.4.3; CIS Benchmarks; PIPEDA s.7
12	Dep. Injection	NIST SA-8; OSFI B-13 s.4.2; ISO 25010
13	Pattern Soph.	NIST SA-8; OSFI B-13 s.4.2; OSFI E-21 s.3; ISO 25010
14	Live Docs	SOC 2 CC2.1; NIST SA-5; OSFI E-21 s.5; ISO 27001 A.5.37
15	SDLC/AI	SOC 2 CC8.1; NIST SA-11/SA-15/SA-17; OSFI E-21 s.4; OSFI B-13 s.4.2; ISO 27001 A.8.25–A.8.31
16	Tech Debt Mgmt	OSFI E-21 s.3; OSFI B-13 s.4.2; NIST SA-22; SOC 2 CC7.1
17	Arch. Phil.	NIST SA-8/SA-17; OSFI B-13 s.4.2; OSFI E-21 s.3; ISO 25010
18	UX from Code	WCAG 2.2; ADA Title III; AODA/IASR; NIST SP 800-63A; SEC Reg S-P; NI 31-103 s.14.2

Validation frameworks: Shared Assessments SIG Questionnaire (21 risk domains, 1,936 questions); FFIEC IT Examination Handbook (11 booklets); OSFI B-13 Technology and Cyber Risk Management (3 domains, 17 principles). Every application-level examination area in all three frameworks maps to at least one of the 18 dimensions.

D Reverse-Mapping Validation: Frameworks to Dimensions

Appendix C maps each dimension to the specific audit frameworks it satisfies. A fair question is whether the 18 industry categories are complete: do they cover what financial institutions actually ask about, or do they select areas

where Application A happens to be strong? The answer can be tested by working in the other direction: starting from the frameworks and mapping their examination areas to the 18 dimensions. If every major examination area maps to at least one dimension, coverage is structural, not selective.

Framework Section	What It Examines	Dimensions
Shared Assessments SIG Questionnaire		
Access Control	Authentication, authorization, privilege mgmt	1, 2, 4
Application Mgmt	Secure SDLC, input validation, API security	6, 13, 15, 18
Asset & Info Mgmt	Data classification, config management	11
Cloud Services	Container security, orchestration	10
Endpoint Security	Service-to-service auth, transport security	3
Info Assurance	Audit logging, evidence preservation	5, 14
Network Security	Segmentation, transport encryption, DDoS	3, 6
IT Operations Mgmt	Change mgmt, monitoring, incident response	7, 8, 9
Operational Resilience	Service continuity, graceful degradation	7, 8, 10
Cyber Incident Mgmt	Detection, logging, alerting, forensics	5, 8
Compliance Mgmt	Regulatory adherence, policy documentation	14, 15
Artificial Intelligence	AI governance, output quality controls	15, 16
Privacy Management	PII handling, data isolation, redaction	4, 5
Threat Management	Vulnerability mgmt, patching, scanning	9, 16
FFIEC IT Examination Handbook		
Information Security	Security program, access controls, crypto	1, 2, 3, 4, 6, 11
Architecture & Ops	IT architecture, infrastructure, operations	7, 8, 10, 12, 13, 17
Dev, Acquisition, Maint	SDLC, change mgmt, testing, QA	9, 15, 16
Audit	Audit function, evidence, compliance	5, 14
E-Banking	Web app security, usability, accessibility	18
OSFI B-13 Technology and Cyber Risk Management		
Principle 4: Tech Arch	Architecture framework, solution design	12, 13, 17
Principle 7: SDLC	Secure development, system maintenance	9, 15, 16
Principle 8: Change/Release	Controlled changes, deployment mgmt	9, 10
Principle 15: Prevent	Multi-layer preventive controls	1, 2, 3, 4, 6, 11

SIG domains that do not appear in this mapping (Human Resources Security, Physical and Environmental Security, ESG, Nth Party Management, Supply Chain Risk Management, Server Security) address concerns outside the scope of application-level software: physical facility access, employee background checks, sub-vendor management, and

hardware configuration. These are organizational controls, not application controls. No software codebase, regardless of quality, can satisfy them; they are addressed through organizational policies and SOC 2 Type II attestation.

Every SIG domain that examines application-level controls maps to at least one of the 18 dimensions. Every relevant FFIEC booklet maps. Every relevant OSFI B-13 principle maps. The 18 dimensions were not selected to favor Application A. They were selected because they are what financial services technology assessments measure at the application level.

References

Akeyless. (2024). State of Secrets Sprawl Report.

Alemohammad, S., et al. (2024). Self-consuming generative models go MAD. *Proceedings of ICLR 2024*. arXiv:2307.01850.

Anthropic. (2025-2026). How AI is transforming work at Anthropic. Internal study, 132 engineers.

Boehm, B. W. (1981). *Software Engineering Economics*. Prentice-Hall.

Brooks, F. P. (1975). *The Mythical Man-Month*. Addison-Wesley.

Capers Jones, T. (2007). *Estimating Software Costs*. McGraw-Hill.

CISQ. (2022). The Cost of Poor Software Quality in the US: A 2022 Report.

Dohmatob, E., et al. (2025). Strong model collapse. *Proceedings of ICLR 2025*. arXiv:2410.04840.

CodeScene. (2024). AI Code Quality Report.

DORA/Google. (2024). 2024 State of DevOps Report.

DORA/Google. (2025). State of AI-Assisted Software Development.

Gartner. (2022). Hype Cycle for Platform Engineering.

GitClear. (2025). AI assistant code quality research: 211 million lines analyzed.

Glassdoor. (2025). Senior Full-Stack Python Developer Salaries, US nationwide.

Glass, R. L. (2003). *Facts and Fallacies of Software Engineering*. Addison-Wesley.

He, J., et al. (2025). Speed at the cost of quality: Quantifying the impact of AI-assisted coding on code quality. arXiv:2511.04427.

ISBSG. International Software Benchmarking Standards Group. Repository of 10,000+ projects.

Lientz, B. P., & Swanson, E. B. (1980). *Software Maintenance Management*. Addison-Wesley.

McKinsey. (2012). Delivering large-scale IT projects on time, on budget, and on value.

McKinsey. (2020). Tech debt: Reclaiming tech equity.

METR. (2025). Measuring the impact of early 2025 AI models on experienced open-source developer productivity.

Mohamed, A., et al. (2025). LLM as a broken telephone: Information distortion accumulates over iterative passes. *Proceedings of ACL 2025*. arXiv:2502.20258.

OX Security. (2025). AI-generated code violates engineering best practices, undermining software security at scale.

MIT Sloan/Hadzima, J. (2005). How much does an employee cost? *MIT Sloan Management*.

Peng, S., et al. (2023). The impact of AI on developer productivity: Evidence from GitHub Copilot. arXiv:2302.06590.

Puppet. (2024). State of DevOps Report.

Ponemon Institute/ServiceNow. (2019). Costs and consequences of gaps in vulnerability response.

QSM/SLIM. Quantitative Software Management. Database of 13,000+ projects.

Shukla, A., Joshi, H., & Syed, A. (2025). Iterative AI code generation and security vulnerability amplification. *Proceedings of IEEE-ISTAS 2025*. arXiv:2506.11022.

Shumailov, I., et al. (2024). AI models collapse when trained on recursively generated data. *Nature*, 631, 755-759.

SonarSource. (2025). State of code quality report.

Stack Overflow. (2025). Developer Survey 2025.

Stripe. (2018). The Developer Coefficient.

Xie, Y., et al. (2025). Experienced developers and AI coding assistants: A longitudinal study of open-source projects. arXiv:2510.10165.