Command Coding: A Methodological Framework for AI-Accelerated Software Development

Thomas S. Villani
thomas.villani@njii.com
New Jersey Innovation Institute
Newark, New Jersey, USA

Martin Kellogg
martin.kellogg@njit.edu
New Jersey Institute of Technology
Newark, New Jersey, USA

Abstract

AI coding agents have created a new paradigm in software development, yet practitioners largely operate without systematic methodology: either using AI for ad-hoc assistance ("vibe coding") or dismissing it as unsuitable for production work. This paper introduces command coding, a structured methodology that applies traditional software engineering management principles, such as architecture, planning, iterative review, and automated quality gates, to the orchestration of AI coding agents. Through an empirical case study of all2md, a production-grade document conversion library supporting 40+ formats (~88,000 code-lines) completed by one senior engineer in 41 working days, we show via triangulated estimation (COCOMO II and Function Point Analysis) that command coding can achieve 25-40× acceleration over traditional development timelines while maintaining production-quality standards.

Command coding amplifies existing expertise rather than replacing it. The methodology requires senior-level skills in architecture, security, and code review, shifting the developer's role from implementer to orchestrator. We propose "contextual quality contagion" as a key mechanism in AI-assisted development affecting code quality: LLMs generate code that matches the quality patterns of their context, creating feedback loops wherein early code quality decisions compound throughout development. This finding suggests that investment in early code quality yields higher returns in AI-assisted projects than in traditional development. Furthermore, the use of automated tooling (linters, type checkers, static analyzers, etc.) within agentic development cycles is critical to avoiding contextual quality contagion.

Keywords

AI-assisted development, software engineering methodology, LLM agents, code quality, human-AI collaboration

> *"A fool with a tool is still a fool. A fool with a power tool is a dangerous, fast fool."*

Conference 2026, City, Country

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The widespread availability of Large Language Model (LLM) coding assistants has fundamentally transformed software development practice. Tools such as GitHub Copilot, Claude, and GPT-based assistants have moved from novelty to ubiquity, with surveys indicating that over 75% of professional developers now use AI assistance in some form [30]. Yet despite this rapid adoption, the industry lacks a coherent methodology for deploying AI in production software development: systems that must be secure, reliable, and maintainable under real-world conditions.

Most practitioners currently use AI coding tools in one of three modes: (1) autocomplete enhancement, treating AI as a smarter IDE feature for line-by-line suggestions; (2) ad-hoc generation, asking AI to produce functions or modules without systematic integration; or (3) autonomous agents, delegating tasks to AI tools that can edit code iteratively within development environments. None of these approaches directly addresses a fundamental question: *Can AI be used to build production-quality software systems systematically, and if so, how?*

The practitioner community has begun developing vocabulary for unstructured AI-assisted development. In February 2025, Andrej Karpathy coined the term "vibe coding" to describe an approach where developers "Accept All always," don't read diffs, and paste error messages back to the AI without analysis, acknowledging this works for "throwaway weekend projects" but not production systems [15]. This informal, iterative style—write a prompt, get code, paste errors back, repeat—optimizes for immediate functionality but produces architectural drift, security gaps, inconsistent quality, and accumulating technical debt; a recent grey literature review found that practitioners frequently experience a "speed-quality tradeoff" [6]. When AI agents observe messy code, they generate more of the same; early shortcuts compound throughout development.

This paper introduces *command coding*, a methodology for AI-accelerated software development that applies traditional software engineering management principles to the orchestration of AI coding agents. The term reflects a shift from developer-as-implementer to developer-as-commander: the human engineer *commands* AI agents through high-level directives while maintaining architectural authority and quality through systematic oversight. Command coding is built on five pillars: 1) human-led architecture, where the engineer defines system structure, patterns, and security requirements before AI implementation begins; (2) structured work decomposition into atomic, independently testable items; (3) iterative multi-model review, where different AI models review each others' output to catch model-specific blind spots; (4) automated quality gates through continuous integration with linting, type

checking, and comprehensive testing; and (5) human oversight at critical decision points including security implementations and API design. The methodology operationalizes the insight that AI coding agents are powerful *implementers* but poor *architects*; by clearly separating these roles, command coding harnesses AI strengths while compensating for known limitations. To assess command coding's effectiveness, we investigate three research questions in the context of a case study:

RQ1 How can LLM-based coding agents be orchestrated systematically to produce production-quality software?

RQ2 What development velocity and code quality can be achieved in a real project using such a methodology?

RQ3 What process mechanisms are critical to mitigating known LLM limitations (e.g., security blind spots)?

Our case study is the development of all2md, a Python library for bidirectional document conversion between 40+ file formats and Markdown, designed for AI/LLM workflows. The project began as a small internal utility (~3,300 lines) and was transformed into a comprehensive library (~88,000 total lines of non-comment, non-blank Python source code, plus a similar amount of test code) over approximately six weeks using the command coding methodology. Properties of document conversion—well-documented formats, clear input/output specifications, pattern-based architecture—are conditions favorable to AI-assisted development, allowing us to explore the methodology's potential. In summary, our contributions are:

- We introduce command coding, a structured development methodology synthesizing software engineering principles with AI agent orchestration that provides a framework for production-grade AI-assisted development (section 2).
- An in-depth exploratory case study of the development of all2md, a production-grade document conversion library supporting 40+ formats completed by one senior engineer in 41 working days, documenting the methodology's application across 504 commits and 150+ review cycles (section 3).
- We analyze the results of our case study; our results suggest that command coding provides acceleration factors of approximately 25-40× compared to traditional development while maintaining high quality standards (section 4).
- We discuss our findings and their implications, and identify *contextual quality contagion* as a mechanism that drives quality in AI-assisted software engineering (section 5).

2 Command Coding Methodology

This section presents command coding independent of any specific project by describing its principles, process phases, strategies for addressing known LLM limitations, and prerequisite skills.

2.1 Overview and Principles

An *AI coding agent* (or just *agent*) is an LLM equipped with tools and operating in a loop (fig. 1). An agent reads code, makes changes, runs tests, observes results, and iterates, much like a human developer. We call an agent's loop with a particular goal a *cycle*: for example, an agent tasked with implementing a feature is in an *implementation cycle*. Command coding is fundamentally about structuring how humans direct and review agent behavior throughout these cycles.

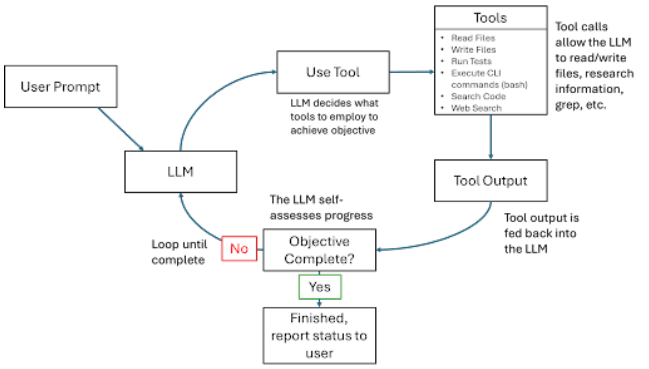

Figure 1: The AI coding agent loop: reading code, making changes, running tests, and iterating.

Command coding operationalizes a key insight: AI coding agents are powerful *implementers* but poor *architects*. By clearly separating these roles of human as architect, AI as implementer, command coding harnesses AI strengths while compensating for its known weaknesses. Command coding has five foundational principles:

(1) **Human-Led Architecture.** The human engineer defines system architecture, design patterns, security requirements, and quality standards *before* the agent generates any code. Design documents serve as a "constitution" guiding all subsequent AI work. Patterns established early influence agents, and so propagate throughout the codebase.

(2) **Structured Work Decomposition.** Complex systems are decomposed into atomic, independently-testable work items with explicit acceptance criteria. Each work item specifies dependencies, expected inputs/outputs, and quality requirements, so that agent sessions have clear success criteria.

(3) **Iterative Multi-Model Review.** Code is reviewed by multiple AI models, wherein a different model than used to generate the code reviews it to catch errors, inconsistencies, and security issues. Different models exhibit different blind spots; cross-model review provides broader coverage than self-review. Reviews occur continuously (every 3–5 work items), not just at project completion.

(4) **Automated Quality Gates.** Continuous integration with linting, type checking, static analysis, and comprehensive testing catches regressions quickly. These tools have two purposes: detecting errors automatically and, through contextual quality contagion (discussed below), signaling to agents that production-quality output is expected.

(5) **Human Oversight at Decision Points.** Critical decisions, such as architecture changes, security implementations, and API design, require human approval. The human engineer monitors agent work and intervenes when agents get stuck, tests pass but the approach seems incorrect, architectural deviation occurs, etc.

Contextual Quality Contagion. We designed command coding around the hypothesis that *contextual quality contagion* is a key driver of the quality of AI-generated code. LLMs generate tokens that fit the statistical distribution of their context. When they observe inconsistent naming, missing type hints, or quick-and-dirty

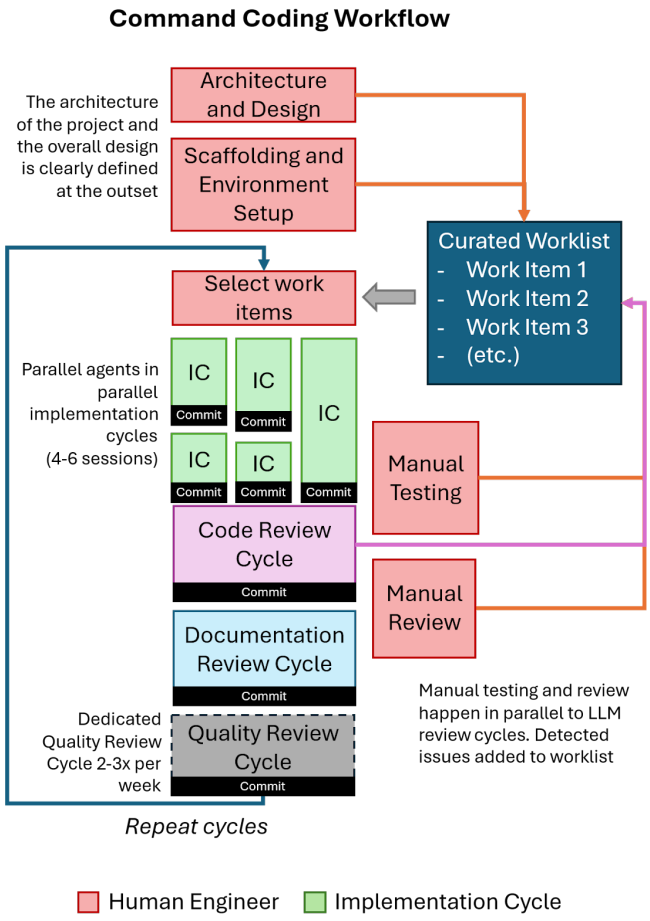

Figure 2: The phases of the command coding methodology.

patterns, they generate more of the same. Conversely, clean, well-documented code with comprehensive type hints and thorough tests increases the probability of clean output. As a result, early code quality decisions compound during AI-assisted development. The first patterns established in a codebase set quality expectations for all subsequent AI-generated code. Early technical debt actively corrupts future generations; early quality investment yields compounding returns. We discuss the evidence we saw in our case study for this hypothesis in section 5.3.

2.2 Process Phases

Command coding follows a four-phase iterative process that parallels traditional software engineering practices (fig. 2). Integration in command coding is continuous. During each phase, version control discipline requires atomic commits with descriptive messages and all commits passing automated quality gates. Documentation updates accompany each significant change: API documentation from docstrings, user guides for new features, and changelog maintenance. The test suite is maintained alongside implementation: new features require new tests, and test failures block integration.

Phase 1: Archtecture, Design, Scaffolding, and Environment Setup (Human-Led). Human engineers produce foundational design artifacts before implementation begins: a system architecture document (component diagram, data flow, technology stack, interface contracts); a pattern library with base classes, configuration patterns, and naming conventions; security requirements including threat model and input validation policies; and quality standards specifying coverage targets, documentation requirements, and code style rules.

Phase 2: Select Work Items (Human-Curated). The design is decomposed using LLMs into a prioritized worklist of atomic work items, reviewed in detail by the human architect. Each item specifies an identifier, description, dependencies, acceptance criteria, complexity estimate, and relevant context files. Prioritization is human-selected, based on architectural considerations such as dependency order (foundations before dependents), parallelizability (independent items can be worked concurrently), and risk (high-risk items come first). The worklist is human-maintained as a living document, updated as work progresses and reviews surface new items.

Phase 3: Implementation Cycles (AI-Executed). For each work item, the AI agent executes within a structured framework. Implementation begins with *context packing*: manually providing the agent with the work item specification, relevant architecture documentation, reference implementations of established patterns, and related test fixtures. The agent then follows a test-driven-development-style [3] implementation loop—writing tests, implementing code to pass tests, running the test suite, analyzing failures, and iterating until all tests pass. Humans intervene when agents appear stuck, when tests pass but the approach seems incorrect, or when a task is deemed by the human architect as security-sensitive.

Phase 4: Code and Documentation Review Cycles. Code review in command coding is continuous, not a single event. *Automated quality gates* run on every commit: linting, type checking, test execution, and security scanning. *Cross-model review* occurs every 3–5 work items: a different AI model than the one that generated the code reviews for correctness, security vulnerabilities, pattern consistency, and architectural alignment. *Human review* focuses on security-critical code paths, API design decisions, architectural alignment, and test adequacy. Review findings, regardless of source, become new work items for the backlog.

In practice, these phases operate in tight daily cycles rather than sequential stages: morning planning of 3–5 work items, daytime AI implementation with human monitoring, afternoon cross-model review, and end-of-day integration with full test suite execution. Larger "holistic reviews" examining entire subsystems occur weekly. Figure 3 shows this *daily sprint* pattern.

2.3 Handling LLM Limitations

Command coding includes mitigations for known AI limitations.

Narrow Context Window. Agents often miss larger patterns in their context, leading to conflicting changes across modules. We mitigate this by splitting discovery and planning from implementation, employing explicit context engineering with relevant patterns, and conducting periodic holistic reviews of entire subsystems.

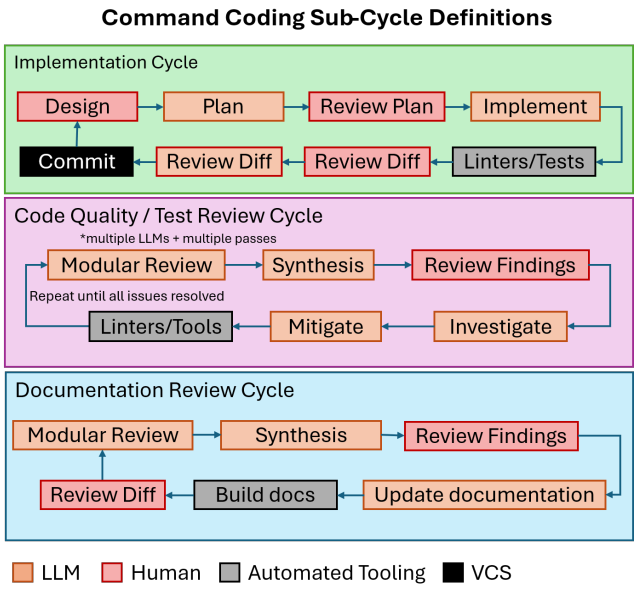

Figure 3: The daily sprint pattern in command coding.

Pattern Deviation Under Complexity. Complex tasks can cause agents to abandon established patterns for seemingly simpler but inconsistent solutions. This is addressed by breaking complex tasks into smaller, pattern-following steps, providing explicit pattern templates in prompts, and reviewing code immediately after complex implementations.

Test-Implementation Confusion. An agent may incorrectly "fix" correct source code to match flawed tests. Mitigation requires human review of test logic, the use of test fixtures with known-correct expected outputs, and prompting the agent to investigate discrepancies before applying fixes.

Security Blind Spots. Agents do not always volunteer security considerations unless explicitly prompted. We require explicit security requirements in every relevant prompt and maintain dedicated security review phases and security-focused test suites.

Additional failure modes, such as version drift toward outdated APIs, accumulating complexity from iterative edits, and hallucinated references to non-existent library signatures, are mitigated through version-pinned prompts, periodic complexity analysis (e.g., `radon`), and strict type checking with `mypy`, respectively. These are less specific to command coding and more general properties of working with LLMs in any agentic context.

2.4 Required Skill Set

Command coding shifts the developer's role from implementer to orchestrator, but this shift requires rather than eliminates deep technical expertise. The methodology demands:

- **Architectural thinking:** Decomposing systems into components, defining interfaces, anticipating integration challenges, and selecting appropriate patterns for the problem domain.
- **Code review expertise:** Rapidly identifying bugs, security vulnerabilities, performance issues, and design flaws in code the reviewer did not write, now applied to AI-generated rather than peer-authored code.
- **Security awareness:** Understanding threat categories, recognizing vulnerable patterns, and explicitly requiring defensive implementations that AI will not volunteer. Special attention to security is especially important given the large number of studies reporting that AI-generated code often contains security vulnerabilities [8, 19, 20, 23].
- **Domain knowledge:** Sufficient expertise to evaluate whether AI output is correct for the specific problem domain, catching plausible-looking but subtly wrong implementations.
- **Test design:** Creating meaningful tests that verify behavior rather than merely exercising code paths, with appropriate use of fixtures, mocks, and edge cases.

These skills are traditionally associated with senior engineers (5–10+ years of experience). A useful heuristic: *if you couldn't effectively lead a team of junior developers building a system, you likely cannot effectively command AI agents to build it.* In fact, we have found that managing AI agents closely resembles managing junior developers: you must provide clear specifications, review carefully, maintain architectural consistency, and handle the hard problems yourself. The key difference is speed: AI implements faster but requires similar oversight effort per unit of code. Section 5.4 explores the implications of this skill shift for software engineering education.

3 Case Study: all2md

This section describes the context, data sources, and methodology of how we applied command coding in the all2md case study. The all2md code is on GitHub[1], and the data on which the conclusions in this section are based is also publicly available, organized into appendices [32].

3.1 Context and Project Overview

all2md is a production-grade Python library for bidirectional document conversion between 40+ file formats and Markdown, designed for AI/LLM workflows. The library provides parsers for diverse input formats (PDF, DOCX, PPTX, HTML, EPUB, EML, RTF, spreadsheets, archives, source code, and more), renderers for multiple output formats (Markdown, HTML, PDF, DOCX, RST, EPUB, and others), and advanced features including an Abstract Syntax Tree (AST) intermediate representation, a transform pipeline for AST manipulation, a plugin architecture via Python entry points, and a Model Context Protocol (MCP) server for LLM integration. The project began as a small internal utility (~3,300 lines of code) supporting a handful of formats with basic conversion capabilities. We used command coding over about 45 working days to transform this small utility into a comprehensive library (over 170,000 total lines of source and tests, excluding comments and blank lines).

The library employs a unified architecture centered on an AST as the intermediate representation for all conversions (fig. 4). Parsers convert input formats to AST nodes; renderers convert AST nodes to output formats. This design enables bidirectional conversion (any supported input to any supported output) and compositional transformations. All 41 parsers follow a consistent pattern: a frozen `dataclass` for options, inheritance from a `BaseParser` abstract

[1] https://github.com/thomas-villani/all2md

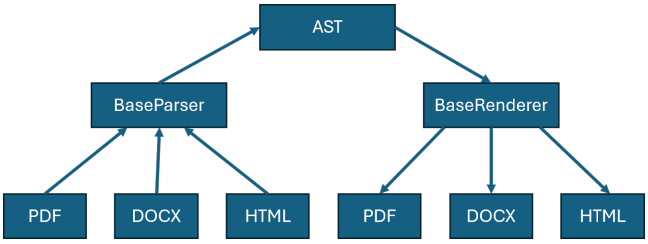

Figure 4: all2md Architecture showing Input Formats → Parsers → AST → Renderers → Output Formats.

class, and a standardized `parse()` method signature. This architectural consistency was established in the first days of development and propagated throughout subsequent implementation.

The all2md project has several characteristics that make it particularly suitable for AI-assisted development:

- **Known document types:** Document formats like PDF, DOCX, and HTML are well documented with public specifications. AI agents have been trained on numerous parser implementations for these formats.
- **Clear architecture:** The parser/renderer pattern with AST intermediate representation is well-established in language and document processing. Once defined, the pattern replicates consistently across 40+ format implementations.
- **Pattern-based development:** The 41 parsers all follow the same structural template. After establishing the `BaseParser` pattern, implementing additional parsers became a matter of format-specific logic within a known structure, where AI agents excel.
- **Testable components:** Document conversion has unambiguous inputs and outputs: a document file in format X produces an AST representation (or rendered output in format Y). This enables automated testing with sample documents and golden/snapshot tests for complex transformations.
- **Greenfield development:** While the project built on a small existing utility, the comprehensive library was essentially greenfield—no legacy constraints, undocumented dependencies, or organizational complexity that might impede AI assistance.

We emphasize that these favorable conditions likely contributed to the results reported; projects without these characteristics may see different outcomes (see discussion in sections 5.2 and 6).

3.2 Data Sources

Our case study draws on these data sources:

- **Version Control History.** Complete git history including 504 commits over 48 calendar days (41 active working days), with timestamps, commit messages, and diff statistics (lines added/deleted per commit). This provides objective timeline data and velocity metrics.
- **Code Metrics.** Static analysis outputs from multiple tools: line counts (total and logical SLOC, production vs. test code), complexity metrics from `radon` (cyclomatic complexity, maintainability index), type coverage from `mypy` (strict mode compliance), and linting results from `ruff`.
- **Test Coverage Reports.** `pytest` coverage reports documenting statement and branch coverage across the codebase, tracked over time to observe coverage progression.
- **Security Analysis.** Results from multiple security scanning tools: Bandit (Python security linter), Semgrep (pattern-based analysis), and CodeQL (GitHub's semantic analysis). Final security posture reflects zero high or critical findings.
- **AI Review Documents.** Over 150 review documents generated during development, capturing cross-model review findings, quality ratings, and identified issues. These documents provide qualitative insight into the review process and issue discovery patterns.
- **Development Notes.** Contemporaneous notes on architectural decisions, pivot points, and observations about AI agent behavior, providing context for interpreting quantitative data.

3.3 Methodology

This study employs a single-case exploratory design following established case study methodology [27, 36]. Single-case designs are appropriate when investigating a phenomenon in depth within its real-world context. The unit of analysis is the all2md project over its approximately six-week development period (September 24 – November 11, 2025), encompassing the complete application of command coding methodology from initial architecture through v1.0 release. We employ mixed methods combining:

- **Quantitative analysis** of development effort (calendar time, working days, commits), productivity (LOC over time, commits per day), and quality outcomes (coverage, complexity, security findings, type coverage).
- **Qualitative analysis** of AI review documents to understand issue discovery patterns, the evolution of quality ratings, and the types of problems caught through multi-model review.
- **Effort estimation** via established parametric methods (COCOMO II) and functional sizing (Function Point Analysis) to enable comparison to traditional, non-AI development.

3.4 Command Coding Instantiation

We employed multiple AI models in coordinated roles. Claude (Sonnet 4 and 4.5) via Claude Code was the primary implementation agent, responsible for writing parser and renderer implementations, following established patterns, and executing test-driven development loops. We used GPT (4.1, 5, 5.1) and Gemini (pro-2.5, pro-3) via direct API for cross-model code review, security analysis, and architectural consistency checking. Claude (Sonnet 4) via Claude Code was responsible for generating docstrings and maintaining documentation. Using different models for implementation and review was deliberate: each model exhibits different blind spots, and cross-model review catches issues that self-review misses.

The project employed production-grade tooling from day one:

- **Linting:** `ruff` with comprehensive rule set including security rules (`flake8-bandit`)
- **Type checking:** `mypy` in strict mode with full type coverage required
- **Testing:** `pytest` with comprehensive markers (unit, integration, e2e, security)
- **Security scanning:** Bandit, Semgrep, and CodeQL

Table 1: Development Summary

Metric	Value
Calendar days	48
Active working days	41
Total commits	504
Production code	~88,000 logical SLOC
Test code	~90,000 logical SLOC
Documentation	~37,000 lines
Parsers implemented	41 formats
Renderers implemented	25+ formats
Test files	233
Review documents	150+

- **CI/CD:** GitHub Actions enforced all quality gates

The project followed a structured review schedule: automated checks on every commit, cross-model AI review every 3–5 work items (approximately twice daily during active development), human review for security-critical code and architectural decisions, and weekly holistic reviews examining entire subsystems. Over the development period, this cadence produced 150+ review documents.

Details on commits, interesting findings from cross-model review, prompt templates, process checklists, and tool configurations in our case study are provided as part of our artifact [32] in appendices A, B, C, D, and E, respectively.

4 Results

This section presents the empirical findings from the all2md case study, addressing development velocity, quality outcomes, effort estimation compared to traditional development, and observed process effects that illuminate critical mechanisms.

4.1 Development Effort and Velocity

Development spanned September 24 to November 11, 2025: 48 calendar days with 41 active working days. Table 1 summarizes the key metrics. The project averaged 12.3 commits per day over the 41 active working days, with a peak of 29 commits on October 9 during a code quality cleanup sprint. Net code addition was approximately 140,000 lines after accounting for ~100,000 lines of refactoring churn. Figure 5 shows cumulative lines of code over the development period, illustrating the progression through four distinct phases:

- **Foundation (Sept 24–27, 65 commits):** Linters, type-checker, CI/CD, core architecture, initial parsers (PDF, DOCX, HTML, EML, RTF, IPYNB), CLI tool, and options system.
- **Format Expansion (Sept 29 – Oct 4, 82 commits):** Archive formats, spreadsheets, additional renderers, and security layer.
- **Advanced Features (Oct 6–13, 147 commits):** AST architecture finalization, transform pipeline, plugin system, and MCP server integration.
- **Polish and Release (Oct 14–26, 210 commits):** OCR support, PyPI release automation workflow, comprehensive documentation, and v1.0 release preparation.

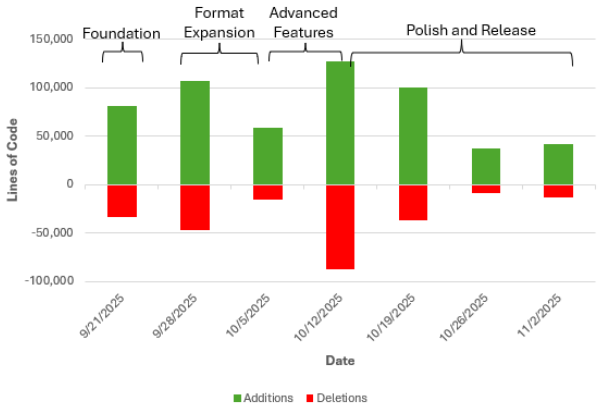

Figure 5: Cumulative LOC over time, showing the Foundation, Format Expansion, Advanced Features, and Polish and Release phases.

Table 2: Code Quality Metrics

Metric	Tool	Result
Test coverage	pytest-cov	87%
Type coverage	mypy (strict)	100% (0 errors)
Lint violations	ruff	0
Security issues	Bandit/Semgrep/CodeQL	0
Cyclomatic complexity	radon	Grade A (avg 5.04)
Maintainability index	radon	Grade A (avg 54.7)
Test-to-code ratio	—	1.02:1

4.2 Code Quality and Security Outcomes

Table 2 presents a suite of code quality metrics from the completed case study. The test-to-production code ratio of 1.02:1 is consistent with ratios observed in TDD-developed projects [17] and exceeds the ratios typical of large-scale open-source repositories [12], reflecting the command coding emphasis on comprehensive test coverage as a quality gate for AI agents.

The library has seen adoption since its public release: as of April 2026, all2md has accumulated over 2,300 total downloads across PyPI and its mirrors, with approximately 433 downloads in the most recent month, reflecting organic adoption without a formal marketing campaign.[2]

4.3 Quantitative Effort Estimation

Estimating traditional development effort for an AI-assisted project presents methodological challenges: there is no counterfactual team that built the same system without AI. We address this through triangulation, using two independent estimation methods and examining where they converge. Table 3 summarizes the estimation methods and their results. Complete details for these estimates are provided in our artifact [32] as appendix F.

COCOMO II Estimation. We applied the COCOMO II Post-Architecture model [5] to the 88,352 logical SLOC of production

[2]Download statistics sourced from ClickHouse ClickPy: https://clickpy.clickhouse.com/dashboard/all2md. As with all PyPI packages, counts include automated systems and CI pipelines and likely overcount distinct users.

Table 3: Effort Estimation Summary

Method	Low Estimate (PM)	High Estimate (PM)
COCOMO II	81	164
Function Point Analysis	40	80
Triangulated Range	**60**	**120**

code. Key parameter selections reflected project characteristics: high precedentedness (document conversion is well-established), very high team cohesion (single developer, no coordination overhead), and very high personnel capability ratings reflecting the senior-level expertise command coding requires. The Effort Adjustment Factor (EAF) of 0.27–0.54 across scenarios yields effort estimates of 81–164 person-months.

Function Point Analysis. We applied IFPUG counting methodology, identifying External Inputs (parsers, input processing), External Outputs (renderers, CLI output modes), External Inquiries (format detection, registry lookups), Internal Logical Files (AST hierarchy, registries, options classes), and External Interface Files (PDF structure, Office XML schemas, etc.). The unadjusted function point count of 663, with a Value Adjustment Factor of 1.09, yields approximately 720 Adjusted Function Points. We applied IFPUG counting methodology, using industry productivity benchmarks of 8–15 hours per function point for Python development [14], yielding 40–80 person-months of traditional effort.

We adopt 60–120 person-months as our triangulated estimate, acknowledging that COCOMO II tends to estimate higher for well-structured projects with experienced developers, while FPA may underestimate effort for libraries with complex internal logic that does not manifest as user-visible transactions. Using this triangulated estimate and conservative actual effort (by treating 41 full working days as 2.0 person-months), we derive acceleration factors. We believe that **approximately 25–40× acceleration** is a defensible, conservative claim that accounts for estimation uncertainty. Even under aggressive downward adjustments (traditional estimate 50% too high, actual effort 50% higher than recorded), acceleration remains substantial (>15×). At a fully-loaded rate of $20,000 per person-month, the triangulated traditional cost range is $1.2M–$2.4M. Actual cost (engineering labor plus ~$550 in AI API costs) was approximately $41,500—a reduction of 96–98%.

4.4 Observed Process Effects

Beyond aggregate metrics, specific episodes illuminate how command coding mechanisms contributed to outcomes.

4.4.1 Establishing Patterns Early. The `BaseParser` pattern, frozen `dataclass` options, and unified attachment handling were established in the first three days. By the end of Week 1, every new parser followed an identical template. This early investment paid compounding returns: implementing a new parser required only format-specific logic, with structure, error handling, and testing patterns automatic. AI agents replicated established patterns with high fidelity, confirming the contextual quality contagion hypothesis—clean, consistent context produced clean, consistent output.

4.4.2 Multi-Model Review. Cross-model review proved essential for catching issues that self-review would miss. Approximately 80% of critical issues were identified by multiple models independently; the remaining 20% were caught by only one model. For example, a September 26 GPT-5 security review identified an SSRF vulnerability that the same-day Gemini review had missed:

> *"SSRF is a real concern... Host checks are naive string checks: only blocks 'localhost', '127.0.0.1', '::1'... Misses 169.254.0.0/16 (AWS IMDS), 100.64.0.0/10... Does not resolve DNS and validate resolved IPs."*

This finding led to comprehensive SSRF protection with IP resolution before requests, redirect target validation, configurable allowlists, and size/timeout limits. The episode illustrates why multi-model review is essential: different models catch different issues.

Review findings also exhibited variability: running the same review prompt twice sometimes produced different findings, with some runs more thorough than others. This variability reinforces the value of multiple review passes and multiple models rather than relying on any single review.

More interesting examples of cross-model review findings are included in the artifact [32] as appendix B.

4.4.3 Where LLMs Failed. Despite overall success, AI agents exhibited consistent failure patterns

- **Test quality issues:** AI-generated test oracles sometimes checked that code did what it did rather than what it *should* do. Tautological tests, excessive mocking, and missing edge cases required humans to review test logic, not just coverage numbers.
- **Complex refactoring:** The AST architecture pivot (October 1–3), affecting 13 converters and 9,500+ lines, required human judgment about risk, timing, and benefit tradeoffs. AI executed the refactoring, but a human decided to proceed.
- **Cascading fixes:** When fixing PDF table detection, an agent modified column detection in ways that broke multi-column layout for documents without tables. CI enforcement (all tests must pass) caught this immediately; the fix required explicit constraints: "ONLY modify table detection logic; DO NOT modify column detection code."
- **Options drift:** Despite established patterns, option naming drifted across parsers (e.g., `attachment_mode` vs. `image_mode` for similar functionality). This accumulated despite systematic review and required a dedicated cleanup pass—illustrating that even with methodology, some inconsistencies slip through.

These failure patterns were *caught* by the process—automated testing, CI enforcement, human review—rather than reaching production. Command coding does not prevent AI errors; it provides systematic mechanisms for detecting and correcting them.

5 Discussion

This section interprets the case study findings in relation to our research questions, examines the conditions under which command coding is most applicable, and elaborates on contextual quality contagion as a hypothesis warranting further investigation. Finally, we discuss implications for practitioners, organizations, and education.

5.1 Research Question Reflections

RQ1: How can LLM-based coding agents be orchestrated systematically to produce production-quality software? The all2md case study demonstrates that the key to systematic orchestration is clear role separation: humans as architects, AI as implementers. The methodology does not reduce oversight burden per unit of code; it increases throughput while maintaining oversight intensity. Production quality comes not from AI capability alone, but from applying software engineering discipline—specifications, review, quality gates—to AI-generated output, much as it does to human-generated output.

RQ2: What development velocity and code quality can be achieved using such a methodology? The case study provides concrete evidence on both dimensions. With respect to velocity, triangulating multiple estimates conservatively puts the acceleration at 25-40×—all while maintaining traditional markers of high-quality code: full type coverage, high test coverage, grade A complexity metrics, and no security warnings from industry-standard static analyzers. These outcomes suggest that the methodology's front-loaded quality investment (architecture, patterns, tooling) yields compounding returns throughout development.

RQ3: What process mechanisms are critical to mitigating known LLM limitations? Three mechanisms were particularly helpful. First, *multi-model review* proved essential for catching model-specific blind spots. The SSRF vulnerability identified by GPT-5 but missed by Gemini's same-day review exemplifies why cross-model validation matters: different models exhibit different failure modes, and no single model provides complete coverage. Approximately 20% of critical issues in all2md were caught by only one reviewing model, validating the redundancy investment. Second, *automated quality gates* served dual purposes: catching errors automatically and signaling quality expectations to AI agents through contextual quality contagion. When agents operate in a codebase with strict linting, comprehensive type hints, and thorough tests, they generate code matching those standards. Without this tooling infrastructure, command coding degrades into vibe coding with extra steps. Third, *early pattern establishment* created the conditions for consistent AI output. The `BaseParser` pattern, frozen `dataclass` options, and unified error handling established in the first week propagated throughout 41 parser implementations. AI agents excel at consistent replication of established patterns; the human contribution is establishing patterns worth replicating.

5.2 Conditions for Applicability

Command coding is not universally applicable. The all2md case study benefited from conditions that favor AI-assisted development, and practitioners should assess whether their contexts share these characteristics. In particular, our application of command coding benefitted from:

- **Pattern-based development** where a structure can be defined once and replicated many times. The 41 parsers in all2md all share a template. Projects with similar replication opportunities—API endpoint handlers, data validators, format converters, test generators—are well-suited to this approach.
- **Well-documented domains** where AI training data includes many similar implementations, like our document-format domain. Web protocols, database schemas, and standard algorithms are other examples of well-represented domains that might benefit from command coding.
- **Clear, testable requirements** that enable tight feedback loops. Document conversion has unambiguous inputs and outputs: a file in format X should produce specific AST nodes or rendered output in format Y. This clarity enables automated testing with ground-truth fixtures and rapid agent iteration against objective success criteria.
- **Modular architectures** with low coupling between components. The all2md architecture allowed parsers to be developed independently; changes to the PDF parser did not affect the DOCX parser.

5.3 Contextual Quality Contagion Hypothesis

We propose *contextual quality contagion* as a mechanism to explain our success with command coding: LLMs generate code that matches the quality patterns of their context, creating feedback loops where early code quality decisions compound throughout development. This mechanism aligns with fundamental LLM behavior: these models generate tokens that fit the statistical distribution of their context. When context includes inconsistent naming, missing type hints, or quick-and-dirty patterns, the probability distribution shifts toward generating more of the same. The mechanism parallels Wilson and Kelling's controversial "Broken Windows" theory [33]—visible disorder signals that standards are not enforced, leading to more disorder. Unlike the controversial proposed behavioral mechanism in humans (that environmental cues influence human choices), the mechanism for LLMs is deterministic: context certainly influences token probabilities.

Several observations from the all2md case study support this hypothesis. When the `BaseParser` pattern was established with comprehensive type hints, detailed docstrings, and thorough error handling, subsequent parser implementations consistently exhibited these qualities without explicit prompting for each. Conversely, when early options classes exhibited naming inconsistencies (`attachment_mode` vs. `image_mode`), these inconsistencies propagated to later implementations despite systematic review—the pattern was replicated faithfully, including its flaws.

The mechanism operates asymmetrically: positive contagion requires active investment (deliberate architecture, early exemplars, enforced tooling), while negative contagion requires only neglect. A single early inconsistency, e.g. an option named differently from its peers, a parser that skips error handling, a test that checks behavior rather than verifying it—becomes load-bearing once downstream agents replicate it. This asymmetry has a practical corollary: the cost of a quality defect introduced in week one of AI-assisted development is not the cost of fixing one instance but the cost of identifying and correcting every instance the agent has silently propagated. Viewed through this lens, the economics of code review shift: early review is not merely good practice but a form of technical debt insurance with unusually high leverage.

Automated quality gates appear to amplify the effect in both directions. When linters, type checkers, and static analyzers are enforced from the first commit, they do more than catch errors: they continuously communicate to the agent what production-quality

output looks like. An agent operating in a codebase where every file passes ‘mypy –strict‘ and every function carries a type-annotated signature will weight its generations accordingly. Conversely, a codebase where quality tools are absent or warnings are routinely suppressed provides no such signal—and the agent has no mechanism to distinguish “this project tolerates slop” from “this project is clean.” This is consistent with the observation in Section 4.4 that the ‘BaseParser‘ pattern, once established with full type coverage, was replicated without explicit re-prompting across all 41 parser implementations.

We propose contextual quality contagion as a hypothesis warranting rigorous validation, not an established finding. A proper validation study would require controlled experiments: identical implementation tasks performed with high-quality context versus low-quality context, with output quality measured against objective criteria. Variables to control include task complexity, model version, temperature settings, and prompt structure. Metrics should include both automated measures (complexity, type coverage, lint violations) and human evaluation of design quality. Such experiments would establish whether the effect is robust, quantify its magnitude, and identify boundary conditions.

The practical implication, if the hypothesis holds, is substantial: investment in early code quality yields even higher returns in AI-assisted projects than in traditional development, where studies have found some “Broken Windows”-like effects [16, 29]. If the first patterns established in a codebase set quality expectations for all subsequent AI-generated code, as the contextual quality contagion hypothesis predicts, it suggests that command coding’s emphasis on production patterns from day one is not merely good practice but a critical success factor for AI-assisted development at scale.

5.4 Implications for Education

The shift from developer-as-implementer to developer-as-orchestrator has implications for how software engineering is taught. Personal programming ability is less important, but engineers using command coding need to be capable in architecture and system design, code review, security thinking, and specification and requirements gathering.

This is a significant inversion of the traditional pedagogical sequence. Introductory software engineering education has long followed a progression from implementation to design: students first learn to write code, then to structure it, and eventually—often only at the graduate level or through industry experience—to reason about systems, interfaces, and quality at scale. Command coding inverts this timeline’s urgency: the skills that previously arrived late (architecture, code review, security thinking, specification) are now the primary bottleneck. A developer who cannot architect a system or evaluate code quality cannot effectively use AI agents to build one, regardless of how capable the underlying model is. The useful heuristic from Section 2.4 bears repeating: *if you could not effectively lead a team of junior developers building a system, you likely cannot effectively command AI agents to build it.* Junior developers and AI agents present similar failure modes—both can produce plausible-looking code that is subtly wrong—and oversight requires the same senior-level judgment in either case. Future curricula should place more emphasis on these senior-level engineering skills and less emphasis on implementation tasks that AI agents can easily complete.

6 Threats to Validity

There are several threats to internal validity. The case study involved only one senior engineer, so it is highly sensitive to that engineer’s skill. The engineer’s effort was also self-reported, which we mitigated by counting their 41 working days conservatively as two person-months. The engineer is also the first author of this study, and so had a motivation to show command coding’s effectivness. We mitigated this threat through: reliance on objective, automated metrics (coverage, complexity, lint violations) that cannot be influenced by interpretation; use of multiple AI models for review rather than self-assessment alone; conservative assumptions throughout effort estimation; and transparent reporting of limitations and failure cases (options drift, test quality issues, cascading fix problems). Git commit history, CI logs, and static analysis outputs provide objective records independent of researcher interpretation, reducing (though not eliminating) the risk that self-reporting biases affect the findings.

Threats to external validity include the fact that this is an exploratory, single-case study; our findings show what is possible under favorable conditions with a specific methodology applied by a specific practitioner; they do not establish what is typical or generalizable. Results may not generalize to other projects, domains, engineers, or AI models. The all2md project was also essentially greenfield, without the constraints typical in legacy systems; command coding may not be applicable to existing projects.

There are threats to construct validity related to our effort estimation techniques and to the chosen code quality metrics. COCOMO II was calibrated on 1990s-2000s projects; modern tooling, languages, and practices may have shifted baseline productivity. The model’s cost drivers involve subjective parameter selection—different analysts might select different ratings. Function Point Analysis was designed for business applications; applying it to libraries requires interpretive decisions about what constitutes a transaction or logical file. We mitigate these concerns through triangulation, conservative parameter selection, and sensitivity analysis. The methods agree within a factor of two, and even aggressive downward adjustments suggest substantial acceleration. Size and industrial complexity metrics imperfectly capture the actual complexity experienced by software engineers [7, 21, 28]. Test coverage measures code execution, not test quality, and can give unwarranted confidence. Type and static analysis coverage does not preclude the presence of other kinds of defects that the chosen analyses do not check for, and practical static analyzers like those that we used can also miss real problems due to analysis unsoundness.

7 Related Work

Industrial Case Studies. The most closely related work to ours are other case studies of coding agents in industry. In a set of case studies at Google, an autonomous agent developed in-house found plausible patches for 73% of machine-reported bugs but only 25% of human-reported bugs, showing the value of human oversight [26]. A case study at ASML, a Dutch photolithography company, found

that smaller, more specialized models had a better cost-benefit tradeoff than larger models, but did not investigate the role of the human engineer as we do [18]. The "State of AI-assisted Software Development," report from Google Cloud [31] further supports our contextual quality contagion hypothesis: they found that AI adoption acts as a "mirror and a multiplier": in cohesive organizations, it boosts efficiency, while in fragmented ones, it highlights and amplifies existing weaknesses.

Productivity Studies. Prior studies on the impact of AI on software engineering productivity have found mixed results. Early experiments found measurable productivity improvements from AI coding assistants [22, 37], though not uniformly [2]. However, a recent study [4] found that experienced open-source developers working on their own repositories were actually 19% slower when using frontier AI tools, despite believing they were faster. A difference-in-differences study examining 807 Cursor-adopting repositories found productivity gains but also potential quality trade-offs, highlighting the complexity of measuring AI's impact on real-world projects [10]. Our command coding case study shows that with the right setup, productivity can be gained without sacrificing quality.

Encouraging Quality. Many recent works have investigated the utility of some of the specific components of command coding for improving the quality of agent-generated code. For example, command coding includes test-driven development (TDD) as a central component; others have also observed that TDD is particularly effective with LLMs [9]. Similarly, other researchers have noted the benefits of automated static analysis feedback [25, 35]; one study in particular specifically showed the benefits of the Bandit analyzer, which we used to analyze all2md, in preventing security vulnerabilities [1]. While these studies validate our use of these tools and techniques, our focus is on the overall framework: command coding could be instantiated with other static analyzers or testing techniques in the future.

Multi-Agent Systems. Recent work has explored multi-agent architectures for software development tasks. Systems like MetaGPT [13] and ChatDev [24] assign distinct roles (product manager, architect, developer, tester) to specialized agents that collaborate through structured communication protocols, though more recent work has questioned the need for such complexity [34]. He et al. [11] provide a systematic review mapping LLM-based multi-agent applications across the software development lifecycle, identifying coordination models (cooperative, hierarchical, decentralized) and communication mechanisms. However, these frameworks typically aim for autonomous operation with minimal human intervention, contrasting with command coding's emphasis on human architectural authority and oversight at decision points.

8 Conclusion and Future Work

This paper introduced command coding, a structured methodology for AI-accelerated software development that applies traditional software engineering management principles to the orchestration of AI coding agents. Through an in-depth case study of the all2md document conversion library, we showed that the methodology can produce production-grade software at substantially accelerated velocity while maintaining quality standards. Contextual quality contagion may (partially) explain our success: establishing a high standard of code quality early induces AI agents to produce more high-quality code, leading to a virtuous positive feedback loop.

Future work should attempt to replicate our results in other domains, with different engineers and/or with teams of engineers, and under more-tightly controlled conditions. Another interesting line of future work is investigating which automated quality gates are most effective at directing AI agents in command coding.

The future of software development is not humans replaced by AI; it is humans collaborating with AI under structured frameworks that leverage AI strengths while compensating for AI limitations. Command coding provides one such framework, grounded in established software engineering principles and validated through empirical application.

Data Availability

The all2md code is available at https://github.com/thomas-villani/all2md, and our data and appendices are publicly available on Zenodo [32].

References

[1] Kamel Alrashedy, Abdullah Aljasser, Pradyumna Tambwekar, and Matthew Gombolay. 2025. Leveraging Static Analysis for Feedback-Driven Security Patching in LLM-Generated Code. *Journal of Cybersecurity and Privacy* 5, 4 (2025), 110.

[2] S. Barke, M. B. James, and N. Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111.

[3] Kent Beck. 2002. *Test-Driven Development: By Example.* Addison-Wesley, Boston.

[4] J. Becker et al. 2025. *Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity.* Technical Report. METR.

[5] Barry W. Boehm, Chris Abts, A. Winsor Brown, Sunita Chulani, Bradford K. Clark, Ellis Horowitz, Ray Madachy, Donald J. Reifer, and Bert Steece. 2000. *Software Cost Estimation with COCOMO II.* Prentice Hall.

[6] Ahmed Fawzy, Amjed Tahir, and Kelly Blincoe. 2026. Vibe Coding in Practice: Motivations, Challenges, and a Future Outlook–a Grey Literature Review. *ICSE-SEIP* (2026).

[7] Janet Feigenspan, Sven Apel, Jorg Liebig, and Christian Kastner. 2011. Exploring Software Measures to Assess Program Comprehension. In *Intl. Symp. on Emp. Soft. Eng. and Meas. (ESEM).* 127–136.

[8] Y. Fu, P. Liang, A. Tahir, Z. Li, M. Shahin, J. Yu, and J. Chen. 2023. Security Weaknesses of Copilot Generated Code in GitHub. *arXiv preprint arXiv:2310.02059* (2023).

[9] Kevin Han, Siddharth Maddikayala, Tim Knappe, Om Patel, Austen Liao, and Amir Barati Farimani. 2026. TDFlow: Agentic Workflows for Test Driven Development. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers).* 1511–1527.

[10] Hao He, Courtney Miller, Shyam Agarwal, Christian Kästner, and Bogdan Vasilescu. 2025. Speed at the Cost of Quality: How Cursor AI Increases Short-Term Velocity and Long-Term Complexity in Open-Source Projects. *arXiv preprint arXiv:2511.04427* (2025).

[11] Junda He, Christoph Treude, and David Lo. 2025. Llm-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Transactions on Software Engineering and Methodology* 34, 5 (2025), 1–30.

[12] Yifeng He, Jiabo Huang, Yuyang Rong, Yiwen Guo, Ethan Wang, and Hao Chen. 2024. UniTSyn: A Large-Scale Dataset Capable of Enhancing the Prowess of Large Language Models for Program Testing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).* doi:10.1145/3650212.3680342

[13] S. Hong, M. Zhuge, J. Chen, et al. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *The Twelfth International Conference on Learning Representations (ICLR).*

[14] Capers Jones. 2008. *Applied Software Measurement: Global Analysis of Productivity and Quality* (3rd ed.). McGraw-Hill.

[15] A. Karpathy. 2025. Vibe Coding. X/Twitter post. https://x.com/karpathy/status/1886192184808149383

[16] William Levén, Hampus Broman, Terese Besker, and Richard Torkar. 2024. The broken windows theory applies to technical debt. *Empirical Software Engineering* 29, 4 (2024), 73.

[17] Lech Madeyski and Marcin Kawalerowicz. 2021. Test-driven development with mutation testing—an experimental study. *Software Quality Journal* 29, 1 (2021), 1–38. doi:10.1007/s11219-020-09534-x

[18] Yash Mundhra, Max Valk, and Maliheh Izadi. 2025. Evaluating Large Language Models for Functional and Maintainable Code in Industrial Settings: A Case Study at ASML. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 3475–3485.

[19] C. Negri-Ribalta, R. Geraud-Stewart, A. Sergeeva, and G. Lenzini. 2024. A Systematic Literature Review on the Impact of AI Models on the Security of Code Generation. *Frontiers in Big Data* 7 (2024), 1386720.

[20] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *IEEE Symposium on Security and Privacy (S&P)*. 754–768.

[21] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. 2021. Program comprehension and code complexity metrics: An fMRI study. In *Intl. Conf. on Soft. Eng. (ICSE)*. 524–536.

[22] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer. 2023. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. *arXiv preprint arXiv:2302.06590* (2023).

[23] N. Perry, M. Srivastava, D. Kumar, and D. Boneh. 2023. Do Users Write More Insecure Code with AI Assistants?. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2785–2799.

[24] C. Qian, W. Liu, H. Liu, et al. 2024. ChatDev: Communicative Agents for Software Development. In *Proceedings of the 62nd Annual Meeting of the ACL*. 15174–15186.

[25] Ravin Ravi, Dylan Bradshaw, Stefano Ruberto, Gunel Jahangirova, and Valerio Terragni. 2025. LLMLOOP: Improving LLM-Generated Code and Tests through Automated Iterative Feedback Loops. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 930–934.

[26] Pat Rondon, Renyao Wei, José Cambronero, Jürgen Cito, Aaron Sun, Siddhant Sanyam, Michele Tufano, and Satish Chandra. 2025. Evaluating agent-based program repair at Google. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 365–376.

[27] P. Runeson and M. Höst. 2009. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering* 14, 2 (2009), 131–164.

[28] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto. 2019. Automatically assessing code understandability. *Trans. on Soft. Eng. (TSE)* 47, 3 (2019), 595–613.

[29] Diomidis Spinellis, Panos Louridas, Maria Kechagia, and Tushar Sharma. 2024. Broken windows: Exploring the applicability of a controversial theory on code quality. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 400–412.

[30] Stack Overflow. 2025. 2025 Developer Survey. https://survey.stackoverflow.co/2025

[31] Google Cloud PDORA Team. [n. d.]. . Technical Report.

[32] Thomas Villani and Martin Kellogg. 2026. Case Study data for all2md. doi:10.5281/zenodo.19686164

[33] J. Q. Wilson and G. L. Kelling. 1982. Broken Windows: The Police and Neighborhood Safety. *The Atlantic Monthly* 249, 3 (1982), 29–38.

[34] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 801–824.

[35] Bo Yang, Jiayi Dang, Huai Liu, and Zhi Jin. 2025. Advancing LLM-Generated Code Reliability: A Hybrid Approach for Hallucination Detection. *IEEE Transactions on Software Engineering* (2025).

[36] R. K. Yin. 2018. *Case Study Research and Applications: Design and Methods* (6th ed.). SAGE Publications.

[37] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian. 2024. Measuring GitHub Copilot's Impact on Productivity. *Commun. ACM* 67, 3 (2024), 54–63.