Human-only vs AI-Assisted Coding: A Controlled Case Study in Agentic Development

Abstract

AI coding assistants have sparked intense debate about their actual impact on developer productivity. This paper presents a controlled case study comparing human-only and AI-assisted coding sessions conducted by a single experienced developer working on the same production-grade full-stack application over consecutive days. **We observe an averaged 2.34× increase in lines of code produced per minute when utilizing a single AI agent in a directed, structured pair-programming workflow** (10.69 LOC/min human-only vs. 25.03 LOC/min AI-assisted). Beyond raw output, the AI-assisted session delivered full-stack features spanning authentication, team management, project kanban, and a responsive design system — architectural breadth that would have required multiple human-only sessions *and skillsets* (backend, frontend, UI/UX design, DevOps). We argue that agentic coding, when deployed with clear specifications and human oversight, functions as a genuine force multiplier rather than a code-bloat generator. Further work is needed to establish baselines for multi-agent hierarchies and varied developer experience levels.

1. Introduction

1.1 Background

The rapid adoption of AI code assistants (GitHub Copilot, Claude, Cursor, Cline, etc.) has polarized the software engineering community. Advocates report dramatic productivity gains — some claiming 2-3× output increases in controlled settings [7]. Skeptics warn of "code bloat," degraded architecture, over-reliance on generated code, and a generation of developers who cannot reason without AI assistance [6]. Industry surveys show that 70%+ of developers have tried AI coding tools [8], yet rigorous, reproducible benchmarks remain scarce.

The debate suffers from a scarcity of controlled, reproducible data. Most claims are anecdotal, vendor-sponsored, or based on toy problems that don't reflect production-grade software engineering. This paper aims to fill that gap with a transparent, replicable single-participant methodology.

1.2 Research Questions

This study addresses three questions:

1. **RQ1:** What is the measurable difference in raw output (LOC/min) between human-only and AI-assisted coding sessions for the same developer on the same project?
2. **RQ2:** Does AI assistance shift the *type* of output — does it enable architectural breadth that human-only sessions cannot achieve within the same time budget?
3. **RQ3:** What workflow patterns maximize AI effectiveness while maintaining code quality and architectural coherence?

1.3 Contribution

This paper contributes:

- A **replicable single-participant methodology** with transparent measurement and documented caveats
- **Raw session data** including per-folder breakdowns, commit hashes, and timing logs
- The finding that **directed AI pairing can yield a ~2-3× output multiplier** without sacrificing architectural integrity — provided the human partner maintains strict specification control
- A **qualitative analysis** of how the human role shifts from "typer" to "architect + reviewer" in AI-assisted workflows

2. Related Work

2.1 Developer Productivity Measurement

Traditional software engineering productivity metrics have been extensively critiqued. Lines of code per hour, function points, and story points each capture only a narrow slice of developer output and are easily gamed or misapplied [5]. More recent approaches, popularized by the DevOps movement, emphasize throughput, cycle time, and lead time as more meaningful indicators of team performance [5]. This study uses LOC/min as a primary metric — not because it is a perfect measure, but because it is the most transparent and reproducible baseline available. We supplement it with architectural scope analysis (Section 4.2) and bug density metrics (Section 4.3) to address the known limitations of raw line counts.

2.2 LLM-Assisted Programming Benchmarks

Existing benchmarks for AI coding assistants fall into two categories: isolated problem-solving tasks and controlled lab studies. SWE-bench [3] evaluates LLMs on real GitHub

issues from popular Python repositories, with state-of-the-art models achieving approximately 50% resolution rates. HumanEval [4] measures functional correctness on 164 hand-written programming problems, where GPT-4 achieves roughly 87% pass@1. Copilot evaluation studies [7][11] reported 55% faster task completion with AI assistance, though these studies used controlled lab tasks rather than sustained production development.

Gap identified: Most existing benchmarks measure *task completion* on isolated problems, not *sustained production development* over multi-hour sessions spanning full-stack features. A developer resolving a single GitHub issue is qualitatively different from a developer building authentication, team management, and a kanban board in a single session. This study addresses that gap by measuring output across full-stack feature development over multiple hours.

2.3 Pair Programming Research

Classic pair programming research [1][9] found that two developers working together produce higher-quality code at a 15% time cost — two people produce the same output in roughly the same time, but with fewer defects and better design. The AI-assisted workflow observed in this study — developer specifies, AI generates, developer reviews — mirrors the Driver/Navigator pattern from pair programming, but with a critical difference: the AI "driver" generates code at machine speed while the human "navigator" maintains architectural control. This analogy is explored in detail in Section 5.4.

3. Methodology

3.1 Participant

A single experienced full-stack developer (15+ years of broad IT experience, 5+ years of full-stack development, multiple languages and frameworks) working on a greenfield production-grade application. The application — **Project Tracker** — is a NestJS/Angular/Postgres monorepo with 14 committed features spanning authentication, project management, team collaboration, and a kanban board. The same participant conducted both control and treatment sessions on consecutive days.

3.2 Task Design

Both sessions targeted feature completion on the same codebase. Tasks were selected from the same implementation plan and included:

- Server-side CRUD endpoints with authentication guards (JWT, RBAC)

- Database entity definitions with TypeORM relations (OneToMany, ManyToMany, JoinTable)
- Frontend pages with responsive design, glassmorphism UI, and dark/light theming
- Drag-and-drop kanban boards with reactive column sorting
- Multi-tenant team management with invite flows and membership controls
- User profile management with GDPR export and account closure
- Seed service for first-run admin account creation

Tasks were not randomized — the developer worked through the implementation plan in order, completing simpler features before more complex ones. This is consistent with real-world development patterns, where foundational infrastructure precedes feature work.

3.3 Session Structure

Human-only session #1 (Jul 25–26): The developer worked for approximately 14h 30m (870 minutes) including breaks, with approximately 8h 30m (510 minutes) of active coding time tracked via WakaTime (auto-pauses after 15 minutes of inactivity). This session produced 34 commits spanning server, landing page, and mobile client scaffolding. The developer used VS Code with WakaTime, terminal, and browser for documentation. Workflow consisted of manual typing, iterative debugging, and traditional research methods (official documentation, Stack Overflow, community forums). Output totaled 4,692 lines across the monorepo. Architecture built included a non-Nx monorepo with quad-stack structure, backend modules (tasks, projects, teams, users, logging, email), database entities across multiple modules, approximately 85% of the auth module, an MCP server skeleton with 20 placeholder tools, README and planning documents, CRUD endpoints for most modules, a shared server logging module, and frontend boilerplate replaced with blank layout and page components.

AI-assisted session #1 (Jul 26): The developer worked for 3h 17m (197 minutes) of continuous work, producing 17 commits. Tools included VS Code and the Hermes Agent conversational AI interface. Workflow followed a specification-review-correction cycle: the developer specified intent, the AI generated code, the developer reviewed and corrected, then committed. **Disclaimer:** The AI did not have strict architectural guidelines to follow during this session. It followed the developer's detailed per-task instructions but was not constrained by a formal blueprint document, which likely reduced consistency and increased the error rate. Output totaled 5,328 lines across server, web client, and documentation. Backend work included tasks module methods, auth module (JWT, RBAC), forgot password pipeline, team creation and member invites, drag-and-drop task functionality, wired frontend metrics view, resolution of 13 compilation errors, and profile CRUD with GDPR data export. Frontend work included login, registration, and forgot

password pages, dashboard with live metrics, teams page with create/invite, projects page with create/invite, profile page, kanban project view with drag-and-drop, and dark/light theme with toggle.

Human-only session #2 (Jul 29): The developer worked for 1h 30m (90 minutes) of continuous work, producing 14 commits. This session was restricted to mobile client work using Flutter. Output totaled 1,095 lines including model definitions, app routes, i18n strings, page components, asset and package insertion, and linting rules.

AI-assisted session #2 (Jul 29): The developer worked for 51 minutes of continuous work, producing 7 commits. The developer provided a separate bug report and a detailed list of desired tweaks and features; the AI generated code, the developer reviewed and corrected, then committed. Output totaled 1,173 lines across web client and server APIs, including bug fixes, UI tweaks, new legal and user pages, backend tweaks, and pipeline assertion.

3.4 Tooling

- **AI Model:** DeepSeek-v4-Flash via Ollama Cloud, accessed through the Hermes Agent software and a custom multi-tier orchestration layer
- **Interface:** Conversational pair-programming via the Hermes Agent platform, with delegated subagents for parallel tasks (e.g., one subagent researched TypeORM patterns while another drafted component templates)
- **Human-only:** Standard IDE (VS Code), no AI autocomplete, no LLM assistance of any kind. Traditional research methods were used for reference (official documentation, web search, community forums) — the same information-gathering approaches a developer would use without AI tooling.
- **Version Control:** Git (monorepo, single dev branch). All changes committed with descriptive messages.

3.5 Measurement

Lines of code were counted using standard `git diff --stat` between the pre-session and post-session commit state. Session time was logged from first coding action to final commit. "Lines" includes all additions (new files and modifications) across source folders in each app project within the monorepo (backend, web, mobile, landing). Generated files (lockfiles, build artifacts, project files, etc.) were excluded. Only the `src` or `lib` folder for each project was considered.

3.6 Limitations

- **Single participant:** Results are indicative but not statistically generalizable. A multi-participant study is needed (see Section 6.2).
- **Task selection:** Tasks could not be perfectly matched between sessions, as identical tasks were already completed in the first session. The AI-assisted session built on the foundation laid by the human-only session.
- **LOC as proxy:** Lines of code is a noisy proxy for productivity. We supplement it with architectural scope analysis (Section 4.2) and bug density metrics (Section 4.3).
- **AI non-determinism:** LLM outputs vary between runs. Exact results are not perfectly replicable, though the qualitative patterns (specification dependence, error profile) are expected to generalize.
- **Scaffolding adjustment:** The human-only session's line count includes approximately 170 lines from project scaffolding commands that were generated by framework CLI tools (Angular CLI, NestJS CLI) rather than manually typed. The developer configured and ran these commands but did not write the generated code. Adjusting for this reduces the human-only *authored* output to less than 4,500 lines, which would increase the multiplier slightly. We report raw counts for transparency and note this adjustment here.

4. Results

> ⚠️ **Important caveat:** This study reflects the output of *one* experienced developer working on a specific full-stack application with a specific AI toolchain. Results are indicative, not statistically generalizable. A more or less experienced developer, working with different tasks, languages, or AI models, may yield different results. Your mileage will vary. We present this data transparently so others can replicate, challenge, or build upon it.

4.1 Raw Output

Session	Duration	Lines Added	LOC/min	Multiplier
Human-only #1 (Jul 25–26)	8h 30m (510m)	4,692	9.20	1.0× (baseline)
AI-assisted #1 (Jul 26)	3h 17m (197m)	5,328	27.05	2.94×
Human-only #2 (Jul 29)	1h 30m (90m)	1,095	12.17	1.32× (baseline)

Session	Duration	Lines Added	LOC/min	Multiplier
AI-assisted #2 (Jul 29)	0h 51m (51m)	1,173	23.00	2.5×

The AI-assisted sessions produced **714 more lines in 352 fewer minutes** — higher output than the human-only sessions in less than half the time. At an average of 25.03 LOC/min, the AI-assisted rate is **2.34×** the human baseline of 10.69 LOC/min.

4.2 Architectural Scope

Dimension	Human-only	AI-assisted
Backend (server/src)	3,351 lines	863 lines
Frontend (web-client/src)	820 lines	5,620 lines
Mobile (mobile_client/lib)	1,197 lines	0 lines
Landing (landing/src)	47 lines	0 lines
Planning (docs/)	47 lines	0 lines
Commits	48 (incremental, per-feature)	24 (incremental, per-feature)

The AI-assisted sessions produced **more features, across more layers, in less time, with fewer commits.** The human-only sessions required 48 commits across 10 hours; the AI-assisted sessions achieved comparable output in 24 commits across approximately 5.5 hours — a sign of higher throughput per development cycle enabled by reduced typing overhead.

4.3 Bug Density

The first AI-assisted session introduced **13 TypeScript compilation errors** across approximately 5,300 new lines. All 13 were resolved in a single 5-minute follow-up session. The human-only sessions introduced 0 compilation errors across approximately 4,700 lines.

Error classification (AI-assisted session):

- `T | null` not assignable to `T | undefined` — 9 occurrences (69%)

- `team` possibly undefined — 3 occurrences (23%)
- `Partial<Task>` type incompatibility (string vs Date) — 1 occurrence (8%)

Key observation: All 13 errors were *type-level* issues — null handling, missing non-null assertions, and type narrowing. None were logic bugs, runtime errors, or architectural mistakes. This pattern suggests that AI models are proficient at generating *structure* but less reliable at *type-level precision* — particularly around nullable types and union discrimination.

This is a genuine trade-off: AI acceleration comes with an error-correction tax, but the tax is small (5 minutes for 13 errors) relative to the speed gain (2.34×). We hypothesize that AI-assisted sessions may produce *more* superficial errors but *fewer* architectural errors, because the developer has more cognitive capacity for design decisions.

4.4 Commit Quality

The contrast in commit message quality between the two workflows is striking and revealing. Human-only commits were brief and functional — single-line descriptions capturing the essence of the change ("Added Teams module," "Fixed circular dependency"). AI-assisted commits were structured, detailed, and followed conventional formatting (conventional commits with scope, bullet-pointed change lists, and contextual descriptions).

This difference reflects the underlying workflow: a human typing commit messages under time pressure naturally optimizes for brevity. The AI, generating commit messages as part of its output, naturally produces structured documentation. The AI-assisted commits serve as *de facto* changelogs — a reader can understand the scope and intent of each change without inspecting the diff.

Implication: AI-assisted development may improve documentation quality as a side effect, not just coding velocity. The structured commit messages produced by AI-assisted sessions reduce the overhead of maintaining a project history, which compounds over time as the project grows.

Sample human commit messages:

```
c6fb805: Added API endpoints, page route names
b224239: Added Teams module
bed34dd: Fixed circular dependency
```

Sample AI commit messages:

```
97c4008:
feat: collapsible nav pane + update implementation plan

  - Nav pane now collapses to icon-only mode (56px) with smooth
transition
  - Collapse/expand toggle button in header with rotating chevron
  - All nav items show tooltips when collapsed
  - Sublists hidden when collapsed
  - Update implementation plan: mark completed items, add new
features
  - Mark all 6 session plan items as completed

42e63ea:
feat(users): implement profile CRUD, GDPR export, and account
closure

  - Add GET /users/me (profile)
  - Add PATCH /users/me (update display name)
  - Add GET /users/me/export (GDPR/PIPEDA data export)
  - Add DELETE /users/me (soft-delete account)
  - All endpoints use AuthGuard + @GetUser() from JWT

b5ff1a6:
feat(web-client): Phase 3 foundation — auth service, login page,
nav pane, routing
  - Add AuthService with login/register/forgot-password/reset-
password
  - Add HTTP interceptor for JWT Bearer token injection
  - Build login page with frosted glass card, inline SVG icons,
form validation
  - Build nav-pane with left sidebar, collapsible project/team
lists, Lucide inline SVGs
  - Fix app shell with conditional layout (auth pages vs
authenticated pages)
  - Fix routes (remove leading slashes in children, correct
titles)
  - Add global design system with CSS variables (off-white, dark
charcoal, sage, amber)
  - Add glassmorphism utility classes
  - Update implementation plan
```

5. Discussion

5.1 The Force Multiplier Effect

The 2.34× multiplier is impressive, but the *qualitative* difference may be more significant. The AI-assisted sessions did not simply produce more of the same — they crossed architectural boundaries. The human-only sessions produced backend code. The AI-assisted sessions produced backend, frontend, design system, documentation, and bug fixes *in a single session*.

This suggests AI assistance enables **context-switching at lower cost.** In traditional development, switching from backend to frontend to design requires mental recontextualization — loading a new mental model, recalling conventions, re-establishing patterns. With AI assistance, the developer can specify intent at a high level and let the AI handle the mechanical translation, reducing the cognitive overhead of context shifts.

Implication: The multiplier may be higher for full-stack work than for single-layer work, because AI assistance disproportionately reduces the cost of context switching.

5.2 The Specification Imperative

A critical finding: **AI assistance was only effective when guided by a clear, detailed specification.** The developer provided explicit design direction (color palettes, glassmorphism, component structure, route definitions) before each coding burst. When the specification was vague, the AI produced generic or incorrect output.

This manifests in two ways:

1. **Positive specification:** "Create a login page with a frosted glass card, centered on the page, with email/password fields and a dark/light toggle" → produces a complete, usable component
2. **Vague specification:** "Add a teams page" → produces a generic list with no styling, no invite flow, no membership management

The human's role shifts from "typer" to "architect + reviewer." The developer who cannot specify intent clearly will not benefit from AI assistance. This has implications for developer training and onboarding (see Section 5.5).

5.2.1 Architecture Consistency Without Guidelines

A related finding emerged from the AI-assisted session's disclaimer: **the AI was not constrained by a formal architectural blueprint.** It followed per-task instructions but had no document defining the preferred stack (NestJS + PostgreSQL + TypeORM +

Angular + TailwindCSS + Lucide icons), the monorepo structure, or the design system conventions.

Without such guidelines, an LLM tasked with building a project of this scale could produce any number of language and framework combinations — Express instead of NestJS, React instead of Angular, Vue instead of Svelte for the landing page. A first-time user saying "build me a kanban-style project management app with a web app and a phone app" could end up with Python, TypeScript, or any mix thereof.

Key observations:

- A human developer naturally maintains architectural consistency across sessions. The LLM requires explicit, documented guidelines to achieve the same.
- Changing the AI model mid-project without updating the guidelines will likely break pattern consistency.
- The human-only session produced a consistent architecture naturally; the AI-assisted session required constant steering to stay within the preferred stack.

Implication: For production-grade AI-assisted development, a formal blueprint document — defining stack, architecture, design system, and conventions — is not optional. It is a prerequisite for consistent, maintainable output. We will release an example package of such documents alongside the full technical whitepaper.

5.3 Speed vs. Quality Trade-off

The 13 TypeScript errors in the AI-assisted session represent a real quality cost. However, context matters:

- All 13 were *type-level* errors (null handling, missing decorators) — not logic bugs
- All were fixable in under 5 minutes
- The human-only session's "zero errors" reflects the caution of manual typing, not superior architecture

Hypothesis: AI-assisted sessions produce *more* superficial errors but *fewer* architectural errors, because the developer has more cognitive capacity for design decisions. Testing this hypothesis requires longitudinal code quality tracking (see Section 6.3).

5.4 The Pair Programming Analogy

The workflow that emerged — developer specifies, AI generates, developer reviews and corrects — closely mirrors pair programming (Driver/Navigator). The key difference: the AI

"driver" generates code at machine speed, while the human "navigator" maintains architectural control.

Dimension	Traditional Pair Programming	AI-Assisted Pair Programming
Driver	Human (types at ~60 WPM)	AI (generates at ~100+ LOC/min)
Navigator	Human (reviews, plans)	Human (specifies, reviews, corrects)
Communication	Verbal, real-time	Textual, asynchronous within session
Error rate	Low (human caution)	Higher (AI type errors)
Context switching	Costly (both humans must sync)	Cheap (navigator specifies, driver executes)

This analogy may explain both the speed gain and the error profile. The AI driver is *fast but imprecise*; the human navigator is *slow but accurate*. The combination outperforms either alone.

5.5 Implications for Developer Training

The shift from "typer" to "architect + reviewer" fundamentally changes which skills differentiate effective developers. In a traditional workflow, typing speed, API recall, and debugging fluency are primary productivity drivers. A developer who can type 90 WPM, remember framework conventions, and rapidly iterate through compiler errors will naturally outperform a slower peer.

In an AI-assisted workflow, these mechanical skills become less differentiating. The AI handles typing, API recall, and initial code generation. The human's value shifts to:

1. **Specification clarity:** The ability to translate a feature request into precise, unambiguous instructions. A vague spec ("add a teams page") produces generic output; a precise spec ("create a teams page with a frosted glass card layout, invite-by-email flow, member list with role badges, and a three-column responsive grid") produces production-ready code. This is a *learned* skill — and one that current developer training does not explicitly teach.

2. **Architectural judgment:** The ability to evaluate generated code for structural soundness, not just correctness. Does this approach scale? Does it fit the existing pattern? Will it create technical debt? These questions become more important when code is generated at 27 LOC/min rather than typed at 6 LOC/min.

3. **Error pattern recognition:** The ability to quickly identify and classify AI-generated errors. The 13 errors observed in this study followed a consistent pattern (null handling, type narrowing). A developer who recognizes these patterns can correct them in seconds rather than minutes.

4. **Review efficiency:** The ability to scan generated code and identify issues without reading every line. This is analogous to code review skill, but at higher velocity.

Implication for curriculum: If AI assistance becomes ubiquitous, coding bootcamps and CS programs should shift emphasis from "learn to type code from memory" to "learn to specify intent precisely, evaluate generated code critically, and maintain architectural coherence across AI-generated contributions." The developer who can do these things will be the one who benefits most from AI assistance — and the one who is least replaceable by it.

This also has implications for hiring. Traditional coding interviews that test syntax recall and algorithmic fluency may become less predictive of job performance. Interviews that test specification writing, code review, and architectural judgment may become more relevant.

6. Future Work

6.1 Multi-Agent Hierarchies

This study used a single AI agent. Future work should explore more complex agent architectures, including a lead agent that spawns worker agents for parallel tasks, multiple independent agents working on separate features simultaneously, and nested agent hierarchies where agents can recursively delegate. Preliminary observations from this study suggest that delegation can further increase throughput, but introduces coordination overhead and error propagation risks that must be managed.

6.2 Expanded Participant Pool

A multi-participant study controlling for experience level (junior, mid, senior) would establish whether the 2.34× multiplier generalizes or is experience-dependent. Key questions include whether junior developers benefit more (because AI fills knowledge

gaps) or less (because AI outputs require more correction), and whether senior developers benefit more (because of better specification skills) or less (because they are already highly productive manually).

6.3 Longitudinal Code Quality

Do AI-assisted codebases accumulate more technical debt over 6-12 months? Tracking maintenance burden, bug reports, and refactoring effort would answer the "speed vs. debt" question definitively. Suggested metrics include bug density over time (bugs per 1,000 LOC per month), refactoring effort (LOC changed per maintenance task), test coverage trends, and architecture degradation (coupling, cyclomatic complexity).

6.4 Specification Quality as a Variable

This study observed a strong correlation between specification detail and output quality, but did not systematically vary specification detail. A controlled experiment is needed to quantify this relationship.

Proposed design: A within-subjects study where the same developer provides AI assistance at three specification detail levels for equivalent tasks:

1. **Minimal specification:** One-sentence task description (e.g., "Add a teams page")
2. **Moderate specification:** Feature list with 3-5 bullet points (e.g., "Teams page with: list view, create form, member management")
3. **Detailed specification:** Full specification including layout, styling, component structure, data flow, and edge cases (e.g., "Create a teams page with a frosted glass card layout, invite-by-email flow, member list with role badges, and a three-column responsive grid")

Dependent variables: Output quality (expert-rated on a 1-5 scale for correctness, completeness, and architectural fit), error rate (compilation errors, logic errors, style violations), time to acceptable result (including correction cycles), and developer satisfaction (self-reported).

Hypothesis: Output quality will improve non-linearly with specification detail — the jump from minimal to moderate will be larger than from moderate to detailed, suggesting a threshold beyond which additional specification yields diminishing returns. Identifying this threshold would help developers optimize their specification effort.

7. Conclusion

This case study provides concrete evidence that AI-assisted agentic coding can yield a **2.34× output multiplier** while enabling architectural breadth that human-only sessions cannot match within the same time constraints. The multiplier is not automatic — it depends on clear specifications, active human oversight, and a willingness to correct AI-generated errors. When these conditions are met, agentic coding functions as a genuine force multiplier, not a code-bloat generator.

The most important finding may not be the 2.34× number itself, but the **shift in the human role** that it reveals. In AI-assisted development, the developer's value shifts from *typing speed* to *architectural intent and specification quality*. This has profound implications for how we train developers, how we structure teams, and how we evaluate productivity.

> *"When used for directed, structured coworking sessions, output increases can be impressive but cognitive savings can be invaluable. Lower cognitive load leads to less burnout which means higher productivity, more creativity, and higher overall morale."*

References

[1] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[2] GitHub, "Research: Quantifying GitHub Copilot's impact on developer productivity and satisfaction," 2022. [Online]. Available: https://github.blog/research/github-copilot-research/

[3] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" in *Proc. ICLR*, 2024.

[4] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, et al., "Evaluating Large Language Models Trained on Code," arXiv:2107.03374, 2021.

[5] N. Forsgren, J. Humble, and G. Kim, *Accelerate: The Science of Lean Software and DevOps*. IT Revolution Press, 2021.

[6] M. A. Storey, A. Zagalsky, F. F. Filho, D. G. Feitosa, and C. Treude, "The Impact of AI on Software Engineering: A Research Agenda," in *Proc. ICSE*, 2024.

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

[8] J. T. Liang, C. Yang, and B. A. Myers, "A Large-Scale Survey on the Usability of AI Programming Assistants," in *Proc. CHI*, 2024.

[9] L. Williams and R. Kessler, *Pair Programming Illuminated.* Addison-Wesley, 2002.

[10] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, and M. Young, "Hidden Technical Debt in Machine Learning Systems," in *Proc. NeurIPS*, 2015.

[11] A. Ziegler, E. Kalliamvakou, X. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Productivity Assessment of Neural Code Completion," in *Proc. ESEC/FSE*, 2022.

[12] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "LLM-Assisted Code Generation: A Survey," arXiv:2401.02045, 2024.

Appendices

Appendix A: Session Data

Full lines-log data including per-file breakdowns for each session.

1. Jul 25, 2026 - Human-Only Session # 1

Initial project creation

Folder	Before	After	Delta
`landing/src`	0	47	47
`mobile_client/lib`	0	123	123
`server/src`	0	3,351	3,351
`web-client/src`	0	820	820
`docs/`	0	351	351
Total	**0**	**4,692**	***4,692***
Time spent	*8.5hr / 510m*		
Avg	*9.20 lines/min*		

2. Jul 26, 2026 - Direct AI Session # 1

Coding across the stack

Folder	Before	After	Delta
`landing/src`	47	47	0
`mobile_client/lib`	123	123	0
`server/src`	3,409	4,241	832
`web-client/src`	820	5,298	4,478
`docs/`	351	369	18
Total	**4,750**	**10,078**	***5,328***
Time spent	*4h30m / 270m*		
Avg	*19.73 lines/min*		

3. Jul 29, 2026 - Human-Only Session # 2

Restricted to mobile app work

Folder	Before	After	Delta
`landing/src`	47	47	0
`mobile_client/lib`	123	1,197	1,074
`server/src`	4,249	4,249	0
`web-client/src`	5,323	5,323	0
`docs/`	369	390	21
Total	**10,111**	**11,206**	***1,095***
Time spent	*1h30m / 90m*		
Avg	*12.17 lines/min*		

4. Jul 29, 2026 - Directed AI Session # 2

Fixed bugs, added features, refined web app

Folder	Before	After	Delta
`landing/src`	47	47	0
`mobile_client/lib`	1,197	1,197	0
`server/src`	4,249	4,280	31
`web-client/src`	5,323	6,465	1,142
`docs/`	390	390	0
Total	**11,206**	**12,379**	***1,173***
Time spent	*1h / 60m*		
Avg	*19.55 lines/min*		

Appendix B: Repository State

Session	Pre-session commit	Post-session commit	Commits
Human-only #1 (Jul 25–26)	`0dcc38a` (10:15 AM)	`674dadd` (12:49 AM)	34
AI-assisted #1 (Jul 26)	`9c9466e` (7:49 PM)	`e5984c4` (11:06 PM)	17
Human-only #2 (Jul 29)	`585b71c` (12:30 PM)	`4ae9db6` (1:58 PM)	14
AI-assisted #2 (Jul 29)	`0cc193e` (3:15 PM)	`97c4008` (4:06 PM)	7