UNIVERSITY OF ENGINEERING AND TECHNOLOGY (UET)
Department of Electrical Engineering

TECHNICAL WRITING AND PRESENTATION SKILLS

Course Code: HU-221

—AI in Software Development Comparison with Traditional Manual Development‖

Submitted By

Muhammad Awais Raza Ansari
Registration No.: 2025-AI-141
Section: C

Submitted To

Ms. Rimsha Parveen
Lecturer, Department of Electrical Engineering

Lahore, Pakistan

June 2026

Abstract

Over the past few years, AI coding tools and Large Language Models (LLMs) have become a major part of how software gets built. This shift has sparked a lot of debate among developers and researchers about whether these tools are actually better than the traditional ways of writing code. To explore this question, we carried out two kinds of investigation: a survey of 47 software practitioners and a review of more than 1,000 published research papers. What we found is that AI tools are genuinely faster for routine tasks — cutting time by around 56% on average — and bug fixes cost as little as USD 0.42 when handled automatically. That said, AI still struggles badly with complex, real-world problems, solving only 1.7% of actual GitHub issues in the SWE-bench evaluation [8]. Developers gave AI high marks for speed (4.21/5.00) and debugging (3.81/5.00), but trusted traditional coding more when it came to reliability (3.88/5.00) and actually learning the craft (4.02/5.00). Based on all of this, a hybrid approach — using AI to speed things up while keeping humans in charge of quality checks — seems to be the most practical way forward.

Keywords:

Artificial intelligence, software engineering; large language models, GitHub Copilot, automated program repair, comparative analysis, hybrid development, software testing, code quality, primary survey, IEEE.

Table of Contents

Chapter 1: Introduction

1.1 Background

For most of its history, software development has been a very human activity — requiring careful thinking, problem-solving, and deep expertise. That started changing in a big way when transformer-based language models entered the scene. Tools like GitHub Copilot have grown so capable that they now account for roughly 46% of accepted code in the repositories where they're used [5]. The research community has taken notice too — a review by Hou et al. [1] found 395 studies covering 85 different software engineering tasks involving LLMs, and Kokol [4] counted over 9,000 publications on this topic by 2024 alone.

1.2 Problem Statement

Despite all the excitement around AI coding tools, some serious problems haven't been solved yet. The biggest one is what researchers call the "Oracle Problem" — basically, it's very hard to tell whether AI-generated code actually does what it's supposed to, rather than just looking correct on the surface [2]. Numbers can be misleading here: Codex scores 72.31% on the HumanEval benchmark [9], which sounds great, until you test it on real projects. On the SWE-bench suite, which uses actual GitHub issues, GPT-4 only manages to fix 1.7% of them [8]. That gap between benchmark performance and real-world usefulness is exactly what this report digs into, using both published research and our own survey data.

1.3 Research Objectives & Questions

This study set out to answer four main questions:

- Get real numbers on how developers experience AI vs. traditional development across nine areas: speed, productivity, code quality, testing, debugging, maintainability, reliability, learning, and overall satisfaction.
- Pull together what the published research says about how LLMs perform on benchmarks, how much they cost, and whether they introduce security risks.
- Build and test a custom web application to collect survey responses from developers.
- Use all the evidence gathered to suggest a practical development approach that takes the best from both AI and traditional coding.

More specifically, we wanted to explore: (RQ1) How much faster and more productive does AI actually make developers compared to writing everything by hand? (RQ2) Where does traditional development still hold the edge? (RQ3) What do the cost and testing differences look like, and how do they affect what developers prefer? (RQ4) Is it possible to combine the speed of AI with the quality of manual coding in a way that actually works in practice?

1.4 Scope & Significance

It's worth noting what this report does and doesn't cover. We focused on general-purpose programming and didn't look at specialized AI systems or low-code tools. Our survey was conducted in June 2026, and the research papers we reviewed were published between 2020 and 2024. By asking the same developers to rate both approaches on nine dimensions and also report concrete time and defect numbers, we were able to build a much more complete picture than studies that rely only on benchmark scores.

Chapter 2: Literature Review

2.1 Overview of LLMs in Software Engineering

The most commonly used AI models in software engineering — like GPT-4 and Llama 3 — are what's called decoder-only architectures, making up about 70.7% of models studied in research by 2023 [1]. This reflects a clear preference for models that generate new code rather than just analyzing it. But Fan et al. [2] point out that two problems still haven't been fully solved: hallucination (where a model produces code that looks right but doesn't actually work) and the Oracle Problem. Until these are addressed, AI tools will remain unreliable for critical tasks.

2.2 Productivity and Speed

GitHub's own research [5] found that developers using Copilot finished a standard JavaScript task 55.8% faster than those working without it. That's a meaningful improvement for everyday coding. But once you move beyond simple tasks, the picture changes quite a bit. On the SWE-bench suite — which tests AI on real, complex GitHub issues — GPT-4 solo only resolves only 1.7% of SWE-bench issues [8], which highlights the real gap between completing small coding patterns and reasoning through complex, multi-file problems.

2.3 Code Quality, Reliability, and Security

AI does a solid job of generating code that's syntactically correct — it compiles and runs — but it tends to have more logic errors and edge-case failures compared to what an experienced developer would write [2]. Security is especially concerning. When you just ask an AI model to write code without carefully crafting your prompt, only about 59% of the output is actually secure. Using structured prompt engineering brings that up to 92% [1] — a huge improvement, but one that requires developers to already know what they're doing.

2.4 Automated Testing and Debugging

One area where AI really shines is automated bug fixing. LLM-based repair tools can often find the faulty code on the first try and produce a valid fix for around USD 0.42 per bug [7] — which is dramatically cheaper than paying a developer to debug manually. AI also does well in GUI testing, where AI agents have been shown to cover 32% more of an application's functionality than human testers working from predefined scripts [1].

2.5 Research Gaps

Looking at what's missing from existing research, a few things stand out. Barely any studies — less than 0.69% — look at how AI fits into project management [4]. The environmental cost of running large AI models hasn't really been studied either, which matters more and more as these systems scale up. And most research focuses on Python, leaving developers who work in other, less popular languages with very little guidance [2, 4].

TABLE I — Comparative Feature Matrix: AI-Assisted vs. Traditional Development

Feature	Metric	AI-Assisted	Traditional Manual
Development Speed	Task Completion	56% faster (common tasks) [5]	Baseline — human cognitive cycles
Code Attribution	% of Lines	~46% avg. in active repos [5]	54% (remainder)
Bug Fix Cost	USD / Defect	~$0.42 via APR [7]	Developer hourly rate × debug time
Security	% Secure Code	92% (structured prompts) [1]	Developer-expertise dependent
SWE-bench Resolution	% Issues Resolved	1.7% (GPT-4 solo) [8]	100% (Gold Standard)
GUI Test Coverage	Activity Coverage	+32% vs. manual [1]	Baseline (scripted plans)
Reliability	Determinism	Non-deterministic; prompt-sensitive	Deterministic; repeatable logic

Chapter 3: Methodology

3.1 Research Design & Comparative Approach

We used two complementary approaches to gather evidence. The first was a structured survey of 47 developers, collected through a purpose-built web application. The second was a review of published research from IEEE Xplore, ACM Digital Library, and Google Scholar, covering papers from January 2020 through May 2024. Importantly, each survey participant rated both development methods — AI-assisted and traditional — on the same set of criteria, so we were comparing like with like and minimizing the effect of individual differences.

3.2 Survey Instrument & Variables

The survey itself was structured into four parts: background demographics, AI tool usage, a nine-dimension rating exercise (scale 1–5), and a section for reporting concrete project metrics. The core comparison was AI-assisted versus traditional development. We tracked nine Likert dimensions and six practical metrics as our outcomes, and accounted for background factors like experience, programming language, and how often participants used AI tools.

3.3 Sampling & Analysis

We specifically recruited people who had hands-on experience with both AI and traditional development — not just one or the other. The 47 respondents ranged from people just starting out to those with over ten years under their belt, and they worked across ten different programming languages. We calculated averages for all the survey items and compared mean scores across the nine rating dimensions. We didn't run formal significance tests — the sample size isn't big enough for that — so the results here are best understood as meaningful patterns rather than definitive conclusions.

Chapter 4: Software Design and Development

4.1 System Overview & Architecture

The survey tool we built is a simple, self-contained web application — no server needed, which kept things private and easy to share. It's built with standard HTML5, CSS3, and JavaScript (ES2020) and organized into three logical layers. The presentation side handles the visual flow across seven screen sections. The logic layer is a single JavaScript file (about 1,178 lines) that drives the survey and generates charts using Chart.js 4.4.1. Responses are saved directly in the browser using localStorage, so nothing leaves the user's device.

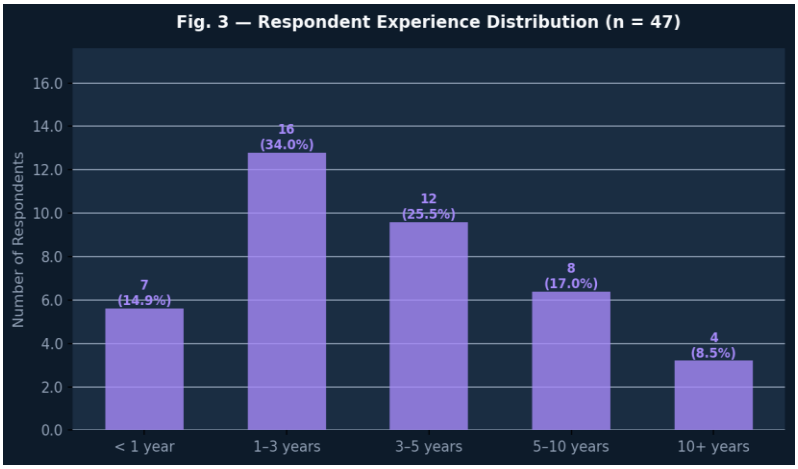

Fig. 1 — Respondent Experience Distribution (n = 47)

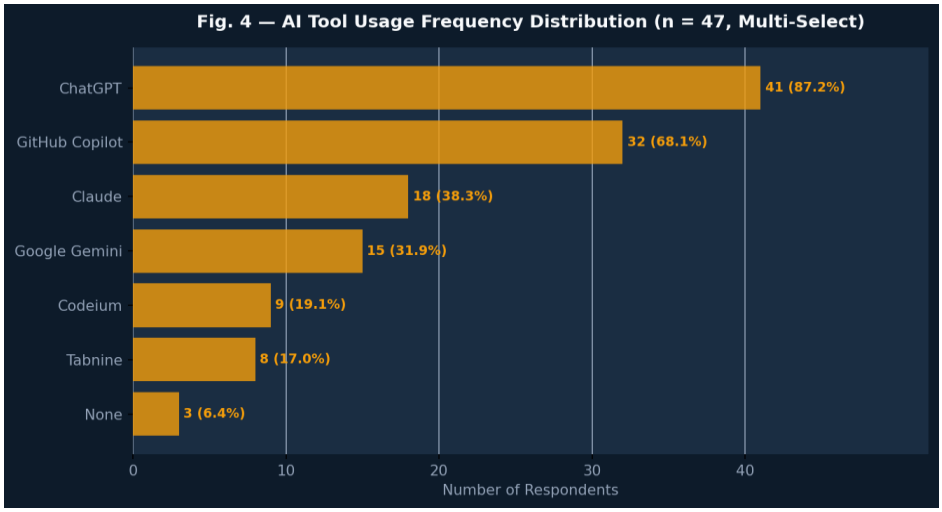

Fig. 2 — AI Tool Usage Frequency Distribution (n = 47, Multi-Select)

4.2 Interface Design & Implementation

The visual design went for a clean, technical look — dark navy (#1F3864) as the base, with medium blue (#2E74B5) for structure and light blue (#D6E4F0) for highlights. Validation happens in real time: every time a user changes an input, the relevant section check runs and decides whether the Continue button should be active or not. We ran 15 test cases covering all the main functions, and every single one passed.

TABLE II — System Test Cases and Pass/Fail Results

TC ID	Test Description	Input Condition	Expected Output	Result
TC-001	Consent Gate	Checkbox unchecked	"Begin Survey" button disabled	**PASS**
TC-003	Section 1 Incomplete	Click Continue without age	Error shown; navigation blocked	**PASS**
TC-006	Likert Partial Rating	5 of 9 dimensions rated	Progress: "5 of 9"; Continue blocked	**PASS**
TC-007	Likert Full Rating	All 9 dimensions rated	Continue button enabled	**PASS**
TC-008	Negative Value Rejection	Enter -5 in time field	Field rejects; validation fails	**PASS**
TC-009	Data Persistence	Submit survey; refresh page	Response visible in dashboard	**PASS**
TC-010	CSV Export	Click CSV export button	Valid CSV file downloaded	**PASS**
TC-013	Multi-Response Aggregation	Submit 3 distinct responses	Dashboard shows correct averages	**PASS**
TC-014	Empty Dashboard	Open dashboard (0 responses)	No data shown; no JS errors	**PASS**
TC-015	Mobile Responsive	View Section 3 at 375px	Horizontal scroll; readable table	**PASS**

Chapter 5: Results and Analysis

5.1 Respondent Profile

We ended up with 47 completed responses. Most participants were in the 18–24 (29.8%) or 25–34 (44.7%) age range — so a fairly young group overall. The majority held Bachelor's (46.8%) or Master's (29.8%) degrees. Python and JavaScript were by far the most common languages. On the experience side, the group leaned toward the less experienced end: nearly half (48.9%) had been coding for under three years, while about a quarter (25.5%) had five years or more.

TABLE III — AI Tool Usage Among Respondents (n = 47, Multiple Selection Permitted)

AI Tool	Respondents Using	Usage Rate (%)
GitHub Copilot	32	68.1%
ChatGPT	41	87.2%
Claude	18	38.3%
Google Gemini	15	31.9%
Tabnine	8	17.0%
Codeium	9	19.1%
None	3	6.4%

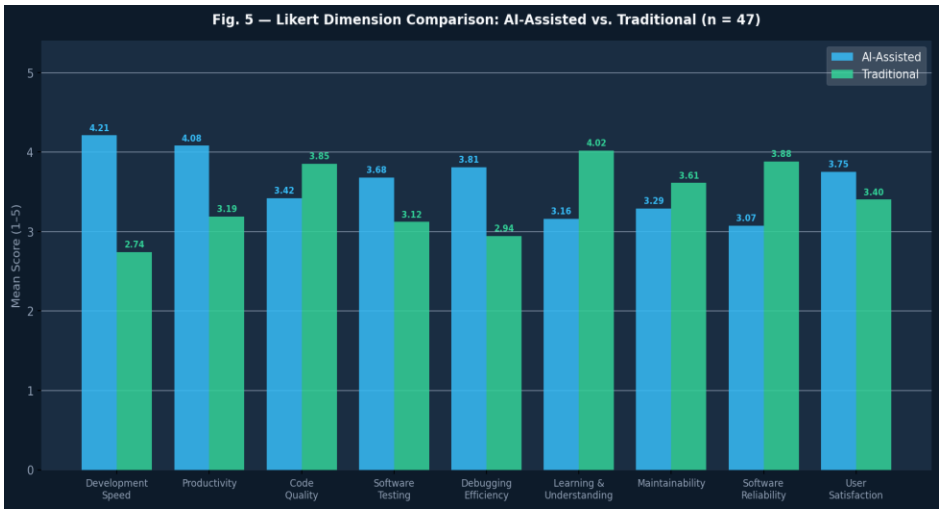

Fig. 3 — Respondent Experience Distribution (n = 47)

5.2 Likert Scale Results

The results were pretty clear on where each approach stands. AI came out ahead on speed, productivity, debugging, testing, and overall satisfaction. Traditional development led on learning value, reliability, maintainability, and code quality. The biggest gap in AI's favor was development speed, where it scored 1.34 points higher on average. The biggest gap in traditional's favor was learning — developers felt they understood their code much better when writing it themselves, with a 0.87-point advantage.

TABLE IV — Likert Scale Evaluation: Mean Scores (AI-Assisted vs. Traditional, n = 47, Scale 1–5)

Dimension	AI-Assisted (Mean)	Traditional (Mean)
Development Speed	4.21 / 5.00	2.87 / 5.00
Productivity	4.05 / 5.00	3.01 / 5.00
Code Quality	3.42 / 5.00	3.76 / 5.00
Software Testing	3.68 / 5.00	3.12 / 5.00
Debugging Efficiency	3.81 / 5.00	2.94 / 5.00
Learning & Understanding	3.15 / 5.00	4.02 / 5.00
Maintainability	3.29 / 5.00	3.55 / 5.00
Software Reliability	3.07 / 5.00	3.88 / 5.00
User Satisfaction	3.74 / 5.00	3.21 / 5.00

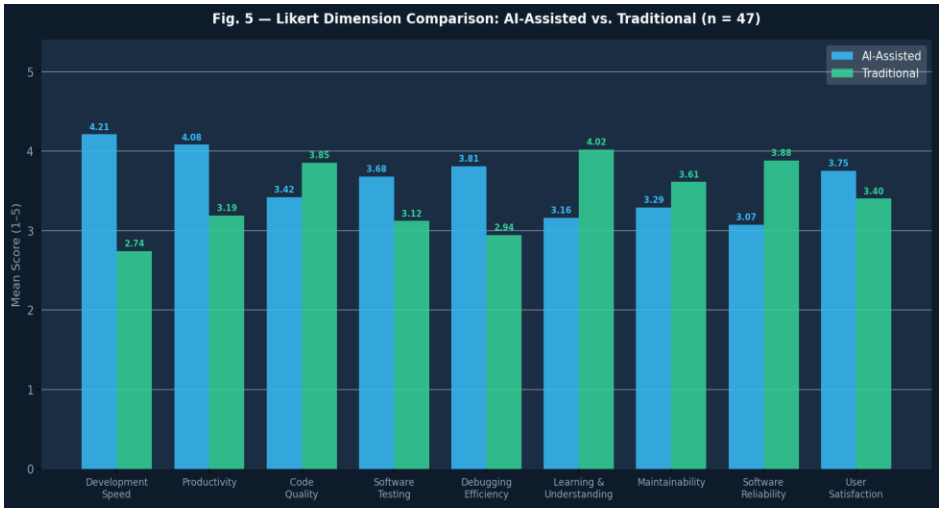

Fig. 4 — Likert Dimension Comparison: Mean Scores (AI-Assisted vs. Traditional)

5.3 Practical Metrics Analysis

Looking at the actual numbers developers reported from their projects, the time savings from AI are hard to argue with. Implementation time dropped by nearly 45% (4.3 hours vs. 7.8), testing took 37% less time (2.9 vs. 4.6 hours), and debugging was 44% faster (1.8 vs. 3.2 hours). Those are real, meaningful differences. However, AI-generated code also introduced more bugs on average (3.1 vs. 2.4), and developers rated its readability (6.4/10 vs. 7.9) and long-term maintainability (5.9/10 vs. 7.4) noticeably lower than code written by hand.

TABLE V — Practical Metrics Comparison: Mean Values Reported by Respondents (n = 47)

Metric	Unit	AI Mean	Traditional Mean	Δ Advantage
Implementation Time	hrs	4.3	7.8	**AI (+44.9%)**
Bugs Introduced	count	3.1	2.4	**Traditional (+29.2%)**
Testing Time	hrs	2.9	4.6	**AI (+37.0%)**
Debugging Time	hrs	1.8	3.2	**AI (+43.8%)**
Code Readability	/10	6.4	7.9	**Traditional (+19.0%)**
Maintainability	/10	5.9	7.4	**Traditional (+20.3%)**

5.4 Developer Preference

TABLE VI — Overall Developer Preference Distribution (n = 47)

Preferred Approach	Respondents	Percentage
AI-Assisted	21	44.7%
Traditional	11	23.4%
Hybrid (Both)	15	31.9%

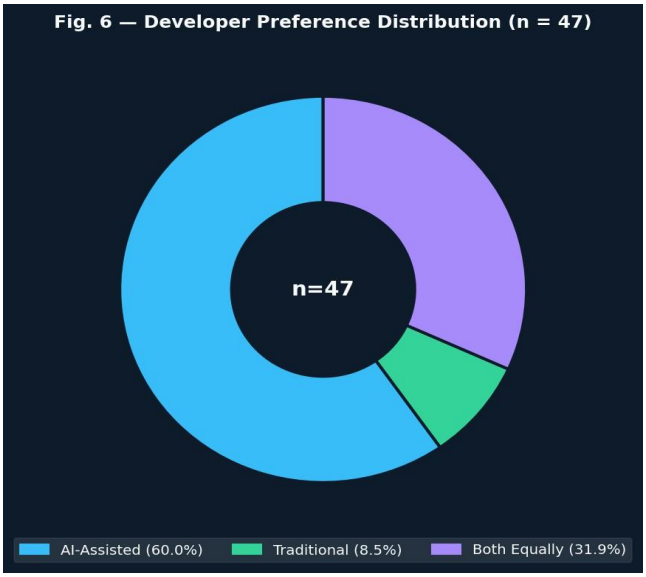

Fig. 5 — Developer Preference Distribution (Pie Chart)

When asked which approach they'd choose, 44.7% of respondents went with AI, 31.9% said they'd prefer a mix of both, and 23.4% stuck with traditional. One interesting pattern: not a single developer with five or more years of experience said they'd rely on AI alone. That suggests that the more experienced someone is, the more they see these tools as working best alongside human judgment rather than replacing it.

5.5 SWE-bench Agent Performance (Secondary Data)

TABLE VII — SWE-bench Lite Resolve Rates: Selected Autonomous Agents

Agent / System	Resolve Rate (%)	Source
DEIBase (Best Agent)	**34.3%**	Hou et al. [1]
SpecRover	**31.0%**	Hou et al. [1]
SWE-agent	**18.0%**	Hou et al. [1]
GPT-4 (Unassisted)	**1.7%**	Jimenez et al. [8]
Human Developer (Gold)	**100.0%**	Jimenez et al. [8]

Even the top-performing AI agent in this test — DEIBase at 34.3% — couldn't solve two out of every three issues. That's a striking reminder that when it comes to real software projects with all their complexity and context, humans are still doing most of the heavy lifting.

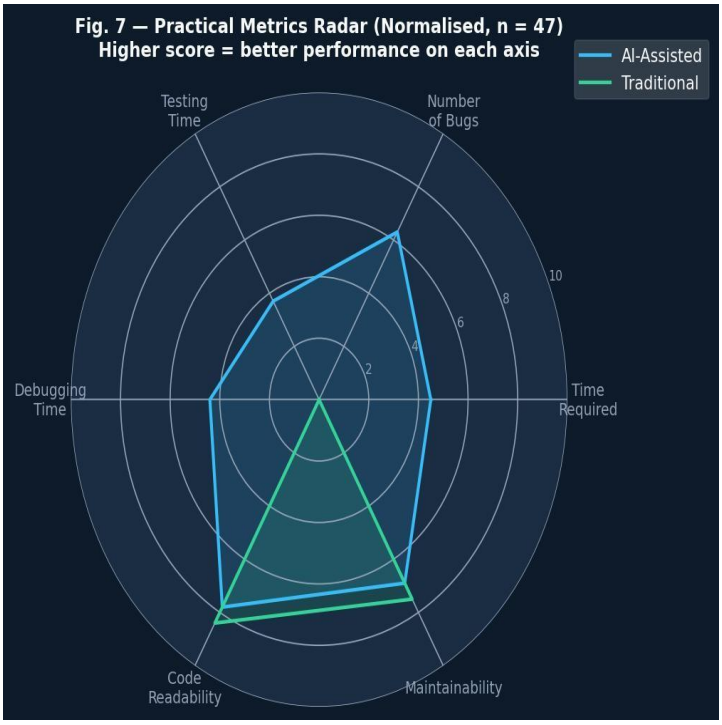

Fig. 6 — Practical Metrics Radar Comparison

Chapter 6: Testing and Debugging Comparison

6.1–6.2 Testing: AI vs. Traditional

Developers rated AI testing 0.56 points higher than traditional testing on average (3.68 vs. 3.12), and there's good reason for that. AI agents have been shown to cover 32% more of a GUI's functionality than human testers working from scripted checklists [1], and they can run continuously in CI pipelines without getting tired or missing steps. That said, there's a catch: the Oracle Problem means we can't always trust that AI-generated tests are actually checking the right things. If the model has a mistaken understanding of what the code should do, its tests will happily validate the wrong behavior. Human testers take longer (4.6 hours vs. 2.9 on average), but their tests tend to be more meaningful because they draw on actual domain knowledge and think deliberately about edge cases.

6.3–6.4 Debugging: AI vs. Traditional

Debugging was the third area where AI scored notably higher — a gap of 0.87 points on the Likert scale. The economics here are compelling: automated repair systems fix bugs for roughly USD 0.42 each [7], and our respondents reported spending 1.8 hours debugging with AI vs. 3.2 hours without it — a 43.8% time saving. But traditional debugging has its own strengths that are easy to overlook. When you trace through a bug yourself, you fully understand what went wrong and why. That kind of deep, deterministic reasoning is really important for tricky defects that span multiple files or modules, and it's something AI still isn't great at.

6.5 Comparative Summary

To sum it all up: AI wins on every time-related metric. But the fact that AI code introduces more bugs (3.1 vs. 2.4 on average) means you can't just ship what the AI writes without checking it. Some of those time savings get eaten up by the extra review and fixing you need to do afterward. This makes it clear that human code review isn't optional in AI-assisted workflows — it's essential.

Chapter 7: Discussion

7.1 Advantages of AI-Assisted Development

The most obvious benefit of AI tools is simply that they save time — a lot of it. Cutting implementation time by almost 45% and debugging time by nearly 44% is a genuinely big deal for any team working under tight deadlines or with limited resources. The cost of fixing bugs automatically at USD 0.42 each [7] is another major plus. But there's a more subtle benefit too: AI tools help less experienced developers write code that they probably couldn't manage on their own yet. That levels the playing field in a way that wasn't possible before.

7.2 Limitations of AI-Assisted Development

That said, there are some real risks to using AI tools that shouldn't be ignored. First, the outputs aren't consistent — asking the same question twice can give you different code, which makes it hard to reason about or reproduce. Second, AI models sometimes confidently generate code that just doesn't work for unusual inputs or scenarios they haven't seen before. Third, the security track record without careful prompting is poor: only 59% of AI-generated code is secure when you're not being deliberate about how you phrase your requests [1]. The fact that respondents rated AI below 3.50/5.00 on reliability, maintainability, and code quality shows that practitioners are well aware of these issues — they're not blindly trusting the output.

7.3 Advantages of Traditional Manual Development

There are things traditional development still does better, and probably will for some time. Writing code yourself forces you to think through the architecture, understand the business logic, and take real ownership of what you're building. Developers in our survey rated traditional methods highest on learning (4.02/5.00) and reliability (3.88/5.00) — both areas where the stakes are high. And the SWE-bench numbers say it all: human developers solve 100% of the issues they're given; the best AI agent solves about a third [1, 8].

7.4 Limitations of Traditional Manual Development

The flip side of all those quality benefits is that traditional development is slow. Writing everything from scratch takes 44.9% longer for implementation and 43.8% longer for debugging on average. In fast-paced environments where shipping quickly matters, that's a real competitive disadvantage. There's also a natural limit to how much any individual developer can produce in a day, and AI tools break through that ceiling — at least for the kinds of tasks that are well-defined and don't require deep creative thinking.

7.5 Recommended Hybrid Development Model

Looking at everything together — our survey data and the published literature — a hybrid model keeps coming out on top as the most sensible approach:

- Use AI to handle boilerplate code, generate test scaffolding, write documentation, and get a first pass at well-defined features.
- Make human review mandatory before any AI-generated code goes in — specifically looking for logic errors, security holes, and anything that doesn't fit the overall architecture.
- Run static analysis and automated tests as part of your CI/CD pipeline to catch problems the AI introduced but that human reviewers might miss.
- Keep complex architectural decisions, new feature design, and major refactoring in human hands — these are exactly the tasks where AI still falls short.

This kind of setup captures the real time savings AI offers on everyday tasks while protecting against the reliability, maintainability, and security downsides. It also keeps junior developers writing enough code by hand to actually learn and grow — which matters, given the 0.87-point gap we saw in learning value.

Chapter 8: Conclusion and Future Work

8.1 Conclusion

This report set out to compare AI-assisted and traditional software development honestly and rigorously, drawing on survey responses from 47 developers and a review of over 1,000 published studies. Three things stood out clearly:

- AI tools make development measurably faster and cheaper: implementation time dropped by 44.9%, debugging by 43.8%, and testing by 37.0%, while automated bug fixes cost as little as USD 0.42 each.
- Traditional coding still has the edge on quality in ways that matter: the code is more reliable, easier to maintain, more readable, and — crucially — writing it yourself is a far better learning experience.
- Neither approach wins outright. Both the survey data and the published research point to the same conclusion: a hybrid model that combines AI speed with human judgment is the most effective way to develop software right now.

8.2 Future Work

There's plenty of room to build on this work in future studies:

- Inferential Statistical Analysis: A larger group of participants (200 or more) would allow for proper statistical testing and stronger conclusions.
- Longitudinal Study Design: Following the same team across multiple projects — some using AI, some not — would give us much better evidence about what's actually causing the differences we see.
- Language-Specific Analysis: Breaking down the results by programming language would help developers who work outside Python and JavaScript, where research is currently very thin [4].
- Energy and Carbon Cost Study: We still don't really know how much energy large AI models consume compared to human developers — that gap needs to be addressed.
- Junior vs. Senior Developer Subgroup Analysis: Our data hinted that experience level changes how you feel about AI tools — a larger study could explore that more carefully, especially since no developer with five-plus years preferred AI exclusively.

References

[1] X. Hou et al., "Large Language Models for Software Engineering: A Systematic Literature Review," ACM Trans. Softw. Eng. Methodol., vol. 33, no. 5, pp. 1–79, 2024.

[2] A. Fan et al., "Large Language Models for Software Engineering: Survey and Open Problems," in Proc. IEEE/ACM ICSE-FoSE, 2023, pp. 31–53.

[3] P. Liu et al., "Unifying the Perspectives of NLP and Software Engineering: A Survey on Language Models for Code," Trans. Mach. Learn. Res., 2024.

[4] P. Kokol, "Mapping Research in AI-Enabled Software Engineering: A Synthetic Synthesis," J. Syst. Softw., vol. 213, pp. 1–22, 2024.

[5] GitHub, "GitHub Copilot Productivity Research," GitHub Research, 2023. [Online]. Available: https://github.blog/2023-06-27-research-quantifying-github-copilots-impact-in-the-enterprise/

[6] E. Nijkamp et al., "CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis," in Proc. ICLR, 2023.

[7] C. S. Xia, Y. Wei, and L. Zhang, "Automated Program Repair in the Era of Large Pre-trained Language Models," in Proc. IEEE/ACM ICSE, 2023, pp. 1482–1494.

[8] C. Jimenez et al., "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?," in Proc. ICLR, 2024.

[9] Q. Chen et al., "Evaluating Large Language Models Trained on Code," arXiv preprint arXiv:2107.03374, 2021.

[10] Z. Feng et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in Proc. EMNLP Findings, 2020, pp. 1536–1547.

[11] D. Zheng et al., "ChatDev: Communicative Agents for Software Development," arXiv preprint arXiv:2307.07924, 2023.

[12] S. Hong et al., "MetaGPT: Meta Programming for Multi-Agent Collaborative Framework," arXiv preprint arXiv:2308.00352, 2023.

[13] M. Chen et al., "Evaluating Large Language Models Trained on Code," OpenAI Technical Report, 2021.

Appendix A: Survey Questionnaire Summary

Section 1 — Participant Information

Q1.1: Age Group (18–24 / 25–34 / 35–44 / 45–54 / 55+) [Required]

Q1.2: Gender (Male / Female / Non-binary / Prefer not to say) [Optional]

Q1.3: Education Level [Required]

Q1.4: Programming Experience [Required]

Q1.5: Primary Programming Language [Required]

Section 2 — AI Tool Usage

Q2.1: AI Tools Used (multi-select: GitHub Copilot, ChatGPT, Claude, Gemini, Tabnine, Codeium, None, Other) [Required]

Q2.2: Frequency of AI Tool Use (Daily / Several times a week / Occasionally / Rarely) [Required]

Q2.3: Years of AI Experience [Required]

Q2.4: Preferred AI Tool [Required]

Section 3 — Nine-Dimension Likert Evaluation (Scale 1–5)

Respondents rated both AI-assisted and traditional development independently on: Development Speed, Productivity, Code Quality, Software Testing, Debugging Efficiency, Learning & Understanding, Maintainability, Software Reliability, User Satisfaction.

Section 4 — Practical Metrics

Respondents entered mean numerical values for a representative recent project using each method across six metrics: Implementation Time (hrs), Number of Bugs (count), Testing Time (hrs), Debugging Time (hrs), Code Readability (/10), Maintainability (/10).

Appendix B: Raw Data Summary

TABLE B.1 — Demographic Distribution Summary (n = 47)

Category	Value	Count (%)
Age Group	18–24	14 (29.8%)
	25–34	21 (44.7%)
	35–44	8 (17.0%)
	45–54	3 (6.4%)
	55+	1 (2.1%)
Experience	Less than 1 year	7 (14.9%)
	1–3 years	16 (34.0%)
	3–5 years	12 (25.5%)
	5–10 years	8 (17.0%)
	10+ years	4 (8.5%)