

From Vibe to Structure: A Pre-Development Framework for AI-Assisted Software Engineering

Muhammad Faiz bin Supian^{1*}, Muhammad Asri bin Muhamad Khidzi¹, Muhammad Alwi Mohd Asmawi²

¹Faculty of Information and Communication Technology, Universiti Teknikal Malaysia Melaka (UTeM), Malaysia

²Faculty of Information Sciences & Engineering (FISE), Management & Science University, Malaysia

*Corresponding Author

DOI: <https://doi.org/10.47772/IJRISS.2026.100800739>

Received: 02 September 2026; Accepted: 07 September 2026; Published: 16 September 2026

ABSTRACT

New AI tools such as ChatGPT and Claude let people build software just by describing what they want in plain language. This way of working is called "vibe coding." A person explains an idea, and the AI writes the code. It is fast, and it is easy to start. It is also very informal. Most developers do not prepare any information before asking the AI to build something. That habit often produces code that misses requirements, contains errors, or does not match what was actually intended. This paper asks a simple question: what should a developer prepare before asking an AI to build software? We propose the Pre-Development AI Context Framework (PDACF), a six-phase framework that guides a developer through project understanding, requirement structuring, system structuring, AI context preparation, AI implementation, and validation, all before any code is generated. The framework builds on established software engineering practice and adapts it to how people actually work with AI coding tools today. To check whether this framework holds up in practice, we surveyed 25 respondents (18 final-year software engineering students and 7 industry professionals) who use AI coding assistants regularly. Each respondent rated their own preparation habits and development outcomes and also reported objective figures such as time spent, number of prompts used, and number of code re-generations. A very strong positive correlation ($r = 0.940$, $p < .001$) emerged between how much a respondent prepared before using AI and how satisfied they were with the outcome. Respondents who prepared well finished tasks 49% faster (1.58 hours versus 3.12 hours). They used 53% fewer prompts and needed 65% fewer code re-generations than respondents who prepared little, a difference that a t-test confirmed was unlikely to be due to chance. These findings are correlational rather than causal, and the sample is small, so the results should be read as a first step rather than a final answer. Even so, they give useful evidence that structured preparation, not only better prompting, deserves closer attention in AI-assisted software development.

Keywords: Vibe coding, AI-assisted software development, prompt engineering, software requirements, human-AI collaboration

1. INTRODUCTION

1.1 Background

AI coding tools have changed how software gets built over the past few years. GitHub Copilot, ChatGPT, and Claude can now write code, explain existing code, find bugs, and suggest designs, all from plain-language instructions [4], [5]. A new style of working has grown out of this capability, often called "vibe coding" [11]. In vibe coding, a developer describes what they want in natural language, and the AI produces most of the implementation. The developer's job shifts. Instead of writing every line, they describe, review, and refine what the AI produces.

This is attractive because it lowers the barrier to building software. A person with limited programming experience can describe an idea and get a working prototype within minutes. That convenience carries a hidden cost, though. Because the process is so informal, many developers simply type a short instruction such as "build a rental management app" and expect the AI to understand everything they have in mind. The AI cannot read minds. When it does not have enough information, it fills the gaps with its own assumptions, and those assumptions are often wrong.

1.2 Research Gap

Traditional software engineering does not work this way. Before any code gets written, a project normally passes through requirement gathering, requirement analysis, system design, architecture planning, and data modelling [1], [2], [3]. These steps exist because they cut down on misunderstanding and rework later. Vibe coding, as it is commonly practised today, skips almost all of it.

Plenty of existing research has looked at how well AI coding tools perform, how to phrase better prompts, and how AI affects developer productivity [6], [7], [8]. Far less has asked a more basic question: what information should a developer prepare before they even start prompting the AI? That is the gap this paper addresses. The issue is usually not that AI cannot write good code. The issue is that AI rarely gets enough context to write the right code in the first place.

1.3 Objectives and Contribution

This paper works toward three goals:

- [1] To identify what information should be prepared before starting AI-assisted software development.
- [2] To organise this information into a structured, easy-to-follow framework.
- [3] To test, through a small survey study, whether this kind of preparation actually improves development outcomes.

To meet these goals, we propose the **Pre-Development AI Context Framework (PDACF)**. PDACF does not replace conventional software engineering. It repackages the useful parts of it into a form that takes little time to prepare and that an AI system can use directly as context. We then test PDACF's core idea with a preliminary survey of 25 respondents, comparing those who prepare thoroughly with those who do not. The paper's contribution is twofold. First, it offers a practical, six-phase framework for preparing AI-assisted software projects. Second, it provides early empirical evidence that preparation level is strongly related to development speed, effort, and satisfaction.

2. PROBLEM STATEMENT AND RESEARCH QUESTIONS

2.1 Problem Statement

Vibe coding makes software development more accessible, but its informal nature can create real problems. Without a clear preparation step, a developer risks sending vague or incomplete instructions that lead to code not matching what was actually needed. Important non-functional needs, such as security, data protection, and compliance, get forgotten. The same request gets repeated many times as code is fixed and re-generated over and over, which quietly cancels out the time AI was supposed to save. The resulting software may not even fit the rest of the system or the organisation's standards.

These risks matter more in fields where mistakes are costly. In a payment system or a healthcare system, missing a security or privacy requirement early on can be very hard to fix later.

2.2 Research Questions

RQ1: What information should a developer prepare before starting AI-assisted software development?

RQ2: How can this information be organised into a structured, reusable framework?

RQ3: Does structured preparation lead to measurably better outcomes, in time, effort, and quality, than unstructured, ad-hoc prompting?

3. THE PROPOSED FRAMEWORK: PDACF

3.1 Six Phases of PDACF

PDACF organises pre-development work into six phases. Each phase produces information that becomes part of the context handed to the AI system before implementation begins.

Phase 1 (Project Understanding). The developer writes down the problem being solved, who the users are, the project's goals, and its scope.

Phase 2 (Requirement Structuring). The developer lists functional requirements (what the system should do), non-functional requirements (performance, security, scalability), business rules, constraints, and acceptance criteria.

Phase 3 (System Structuring). The developer sets out the user flow, the data model, a high-level architecture, the main modules, the technology stack, and any external dependencies.

Phase 4 (AI Context Preparation). The developer packages all of this into a form the AI can use directly: project context, coding guidelines, the AI's role and boundaries, expected output format, and validation rules.

Phase 5 (AI Implementation). With this context ready, the developer sends structured prompts to the AI, receives the generated code, reviews it, and gives feedback.

Phase 6 (Validation). The output is checked against requirement coverage, functional completeness, code quality, architectural consistency, security, and maintainability. Where something is missing, the developer goes back to an earlier phase to fix the context, not just patch the code.

Figure 1 sets out this flow.

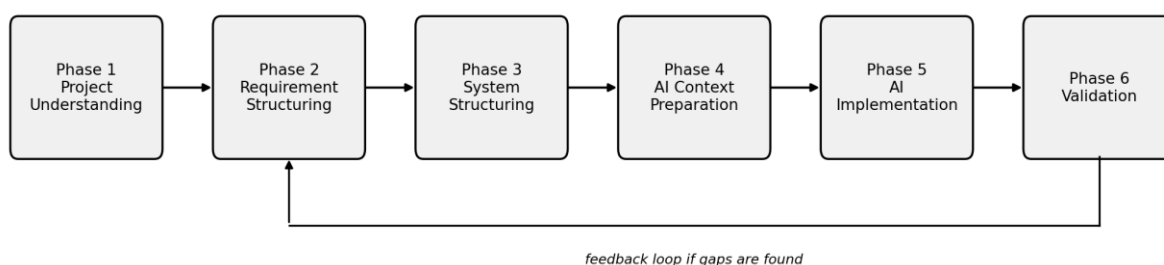


Figure 1. The PDACF pipeline. Project Understanding leads to Requirement Structuring, then System Structuring, then AI Context Preparation, then AI Implementation, then Validation, with a feedback loop from Validation back to the earlier phases whenever gaps turn up.

3.2 What Makes PDACF Different

Most existing guidance on working with AI coding tools focuses on prompt wording: how to phrase a single instruction well [6]. PDACF takes a step back from that. Rather than asking "how do I write a good prompt," it asks "what do I need to know and prepare before I write any prompt at all." The focus moves from prompt quality to context completeness, a related idea but a distinct one. PDACF also keeps a place for security, ethics, and compliance from the first phase onward, rather than treating them as something to add later. This matches long-

standing advice in the software engineering literature, even though that advice is often skipped in informal AI-assisted workflows [1], [12], [13].

3.3 Comparison with Existing Approaches

PDACF does not stand in a vacuum. Several existing bodies of work already touch parts of what it covers, and it is worth being explicit about where PDACF overlaps with them and where it adds something new. Table 1 lays out this comparison across five relevant strands of prior work.

Table 1. PDACF Compared with Existing Approaches

Approach	Main Focus	Stage Covered	Security/Compliance	Gap PDACF Addresses
Prompt engineering patterns [6]	Wording and structure of a single instruction	Implementation only	Not addressed	Describes how to phrase one prompt well, not what a developer should prepare before any prompt is written
LLM-assisted requirements elicitation [9], [10]	Using an LLM to help draft or analyse requirements	Requirements phase only	Not addressed	Treats requirements gathering as something the AI helps do, rather than context the developer prepares beforehand
Conventional SE lifecycle guidance [1], [2], [3]	Full lifecycle: requirements, design, build, test	All phases, written for human-only teams	Addressed generically, not AI-specific	Not adapted to a conversational, prompt-driven implementation step
LLM-for-SE surveys [7], [14]	Cataloguing the risks and open problems of using LLMs in software engineering	Broad and descriptive	Flags the risk, does not propose a fix	Names problems such as insecure defaults and hallucinated logic, but stops short of a preparation workflow
Security-by-design principles [1], [12], [13]	Building security and ethics considerations in from the start	Design phase	Central focus	Well established in principle, but not tied to an AI-prompting workflow specifically

The pattern across these strands is that each one covers a slice of the pre-development problem, prompt wording, requirements, general process, known risks, or security, but none combines them into a single, sequential checklist aimed specifically at the moment before a developer opens an AI coding assistant. That is the specific

gap PDACF is built to fill. It is best read as an integration of existing, individually well-supported ideas into a workflow shaped for AI-assisted development, rather than as a wholly new concept invented from nothing.

4. METHODOLOGY

4.1 Research Design

This study follows a mixed-methods design carried out in two stages. Stage 1 built PDACF from a review of established software engineering practice [1], [2], [3] and from the authors' own experience running AI-assisted software projects. Stage 2 tested PDACF's core idea, that preparation level matters, through a descriptive survey of 25 respondents. We present this paper as a preliminary study. Its aim is to offer early, honest evidence rather than a final, statistically definitive answer, and Section 6.5 discusses its limitations openly.

4.2 Participants and Recruitment

Twenty-five respondents took part: 18 final-year software engineering students with at least six months of hands-on experience using AI coding assistants, and 7 industry professionals with two to five years of experience using AI-assisted development on real projects. Students were recruited through final-year-project cohorts at the authors' home institutions and invited by email and class announcements; industry professionals were recruited through the authors' professional networks. Participation was voluntary and unpaid. All respondents had used tools such as GitHub Copilot, ChatGPT, or Claude for at least three months, and each had completed at least one software feature with AI assistance shortly before taking the survey. This convenience-sampling approach was chosen for reach within the study's short timeframe, and its effect on how far the findings can generalise is discussed in Section 6.5.

Respondents were split into two groups based on their median preparation score, calculated from five Likert questions (Q1 to Q5, described below):

- **High Preparation (n = 13):** preparation score ≥ 3.80
- **Low Preparation (n = 12):** preparation score < 3.80

The preparation score for each respondent was the arithmetic mean of their Q1 to Q5 ratings, each on a 1-to-5 scale (for example, a respondent scoring 4, 4, 3, 4, 3 on Q1 to Q5 would receive a preparation score of 3.6). The cut-off of 3.80 is the sample median, so the split is a within-sample comparison rather than a fixed external benchmark; a different sample would likely produce a different cut-off value.

The tasks respondents worked on were not assigned by the study and varied by respondent. Based on the short written descriptions collected alongside the survey, the most common task types were small backend features (database queries, API endpoints), user-facing form and validation logic, and minor UI or integration fixes, the kind of single-feature work a student or junior developer typically completes with an AI assistant in a few hours. Task complexity and scope were not held constant across respondents, which is treated as a limitation in Section 6.5 rather than a controlled variable.

4.3 Data Collection Instrument

Data was collected through a two-part questionnaire, developed specifically for this study and mapped directly onto the phases of PDACF (Section 3.1). Draft items were first written to correspond to each PDACF phase, then reviewed internally by the author team for clarity and redundancy, and revised into their final wording before distribution; no separate pilot sample was run ahead of the main survey, which is acknowledged as a limitation in Section 6.5.

Part A (Perception questions, Q1 to Q10). Ten questions used a 5-point Likert scale, from 1 (Strongly Disagree) to 5 (Strongly Agree). Q1 to Q5 measured preparation habits: whether the respondent defined requirements, provided context, broke tasks down, planned the technical setup, and set coding rules before

prompting the AI. Q6 to Q10 measured perceived outcomes: development speed, how often re-prompting was needed, whether the AI's output matched requirements, whether testing was done, and overall satisfaction.

Part B (Verification questions, Q11 to Q15). Self-reported opinions can run optimistic, so five further questions asked for concrete, factual answers instead of ratings. Respondents listed exactly what information they gave the AI, how they structured their prompts, and which technical decisions were made in advance. They also reported objective numbers: actual hours spent with and without AI, the number of prompts used, and how many times code had to be regenerated or fixed. This part asked respondents to classify their final outcome into one of five categories too, ranging from "worked correctly on the first try" (A) to "had major problems and needed to be rebuilt" (D), with a fifth category for "no specific testing was done" (E).

This two-part design let us check whether a respondent's self-rated satisfaction matched what they actually reported doing, which cuts down the risk of the results resting on overclaiming alone.

4.4 Data Analysis

The analysis proceeded in four steps. First, descriptive statistics (mean, median, standard deviation) were calculated for each Likert question. Second, Pearson correlation was calculated between the preparation score (Q1 to Q5) and the effectiveness score (Q6, Q8, Q9, Q10), alongside the p-value, a 95% confidence interval (via Fisher's z-transformation), and the coefficient of determination (r^2) for each correlation. Third, the high-preparation and low-preparation groups were compared on objective measures such as time, number of prompts, and number of code re-generations, using an independent-samples t-test and Cohen's d to report the size of each difference, not only its direction. Fourth, the short written answers describing what went right or wrong in each respondent's most recent AI-assisted task were reviewed thematically.

Likert-scale data is ordinal rather than strictly interval, which is a standard caveat for Pearson correlation on this kind of data. To check that the reported relationships were not an artefact of that assumption, Spearman's rank correlation (which does not assume interval spacing) was also calculated for each pair of variables as a robustness check; the results are reported alongside the Pearson figures in Section 5.2. With only 25 respondents, all inferential statistics in this paper should be read as indicative rather than conclusive, and none of the reported correlations should be interpreted as evidence of a causal relationship. Section 6.1 and Section 6.5 return to this point directly.

5. RESULTS

5.1 Descriptive Statistics

Table 2 shows the average score for each of the ten Likert questions.

Table 2. Descriptive Statistics for Q1–Q10

Question	Mean	Median	Std. Dev.	Dimension
Q1	3.84	4.0	0.94	Requirement preparation
Q2	3.96	4.0	0.84	Context preparation
Q3	3.68	4.0	0.95	Task breakdown
Q4	3.72	4.0	0.94	Technical planning
Q5	3.56	4.0	1.00	Development constraints
Q6	4.32	4.0	0.63	Development speed

Q7	3.28	3.0	1.10	Iteration effort (reverse-scored)
Q8	3.68	4.0	0.90	Requirement adherence
Q9	4.20	4.0	0.76	Validation
Q10	4.12	4.0	0.78	Overall benefit

Note: Q7 is reverse-scored. A lower score here means fewer iterations were needed, which is the better outcome. The average preparation score (Q1 to Q5) came to 3.75 out of 5.0, and the average effectiveness score (Q6, Q8 to Q10) came to 3.92 out of 5.0. Preparation scores varied more between respondents than effectiveness scores did (standard deviation as high as 1.00), which suggests that without a shared framework, people prepare very differently before turning to AI.

5.2 Correlation Findings

Three correlations stood out. For each one, we report the Pearson coefficient alongside its p-value, a 95% confidence interval, the coefficient of determination (r^2), and the Spearman rank correlation as a robustness check against the ordinal nature of Likert data (see Section 4.4).

Finding 1: Preparation and satisfaction are strongly related ($r = 0.940$, $p < .001$, 95% CI [0.868, 0.974], $r^2 = 0.884$; Spearman $\rho = 0.943$, $p < .001$). Respondents who prepared more thoroughly reported markedly higher satisfaction with the overall development process. The r^2 value indicates that preparation score alone statistically accounts for around 88% of the variance in effectiveness score within this sample.

Finding 2: Preparation and iteration effort are strongly, negatively related ($r = -0.837$, $p < .001$, 95% CI [-0.926, -0.659], $r^2 = 0.700$; Spearman $\rho = -0.819$, $p < .001$). More preparation went with fewer repeated prompts, reflected in lower Q7 scores.

Finding 3: Preparation and the actual number of prompts used are also strongly, negatively related ($r = -0.944$, $p < .001$, 95% CI [-0.975, -0.875], $r^2 = 0.891$; Spearman $\rho = -0.944$, $p < .001$). This lines up with Finding 2, but using objective, self-reported numbers rather than opinions. High-preparation respondents needed an average of 5.2 prompts, against 11.1 for low-preparation respondents, a 53% reduction.

The close agreement between each Pearson and Spearman figure suggests the relationships are not simply an artefact of treating ordinal Likert scores as interval data. These three results point the same way: better preparation goes with less back-and-forth and higher satisfaction, and that shows up both in how people felt about the process and in how many times they actually had to return to the AI. That said, a correlation this strong in a sample of 25 self-selected respondents should be read as a strong association rather than proof that preparation causes the outcome; Section 6.1 returns to this distinction.

5.3 Objective Behavioural Data

Table 3 summarises the objective data respondents reported about their most recent AI-assisted task.

Table 3. Development Time, Prompts, and Re-generation

Metric	Mean	Median	Range	Std. Dev.
Time with AI (hours)	2.32	2.50	1.0–4.5	1.10
Estimated time without AI (hours)	4.28	4.00	3.0–6.0	1.05

Time saved	1.96 hrs (45.8%)	1.50 hrs (37.5%)	15%–80%	0.85
Number of prompts	8.0	8	3–16	3.15
Number of re-generations/fixes	3.6	3	1–9	2.10

On average, using AI saved respondents about 1.96 hours, or 45.8%, against their own estimate of how long the same task would take without AI. That benefit varied a great deal between respondents, though, and an average of 8 prompts per task shows that AI-assisted development in this sample stayed an iterative, back-and-forth process rather than a single-shot generation.

5.4 High vs Low Preparation Comparison

Table 4 compares the high-preparation group ($n = 13$) and the low-preparation group ($n = 12$) directly.

Table 4. High vs Low Preparation Groups

Metric	High ($n=13$)	Low ($n=12$)	Difference
Development time	1.58 hrs	3.12 hrs	-49%
Number of prompts	5.2	11.1	-53%
Number of re-generations	1.9	5.4	-65%
Q7 iteration score	2.08	4.58	-55%

The high-preparation group finished the same kind of task in roughly half the time the low-preparation group needed, using about half as many prompts and far fewer re-generations. An independent-samples t-test on the number of prompts confirmed this difference is unlikely to be due to chance alone ($t(23) = -6.13$, $p < .001$), and the effect size was large (Cohen's $d = 2.51$). Because these are the same 25 respondents split by their own preparation habits, not two separate populations, the gap between the groups is a useful signal even in a sample this small, though it should still be read as a between-group difference rather than a controlled experimental effect, since the two groups were not randomly assigned.

5.5 Final Outcome Categories

Respondents also classified how their most recent AI-generated feature turned out, from working correctly right away (A) to needing a full rebuild (D), or going untested altogether (E). Table 5 shows the distribution and the preparation score tied to each outcome.

Table 5. Final Outcome Categories

Category	Description	Count	%	Avg. Preparation Score
A	Met requirements with no major changes	6	24%	4.70
B	Needed a few small changes	8	32%	4.15
C	Needed several rounds of debugging	7	28%	3.42
D	Had major problems, needed rebuilding	3	12%	2.67

E	Was not tested specifically	1	4%	—
---	-----------------------------	---	----	---

The gap between the best outcome (A, 4.70) and the worst (D, 2.67) comes to 2.03 points on a 5-point scale, a large and consistent gap. Respondents who landed in Outcome D often said they gave the AI "just a rough idea" and hoped it would understand what they meant. Respondents who reached Outcome A typically reported spending 30 to 45 minutes on planning before writing their first prompt.

6. DISCUSSION

6.1 What the Strong Correlation Means

The very strong relationship between preparation and satisfaction ($r = 0.940$) backs the central idea of this paper: preparing project information before the first AI prompt is not a minor detail. It is closely tied to how the whole process turns out. That does not mean prompt wording is unimportant [6]. It means that what the AI is told, before the conversation even starts, may matter at least as much as how it is told. Put simply, the old computing saying "garbage in, garbage out" turns, in this setting, into something closer to "clarity in, clarity out."

This relationship is correlational, not causal, and should be read that way. The survey did not manipulate how much any respondent prepared; respondents simply reported their own habits and their own outcomes side by side. A third factor, such as a respondent's general experience level or familiarity with the problem domain, could plausibly drive both variables at once: a more experienced developer might simply prepare more and work more efficiently for reasons that have little to do with preparation itself. What the correlation shows is that preparation and positive outcomes travel together closely in this sample. It does not, on its own, prove that preparing more would improve outcomes for any given individual developer. Section 6.5 returns to what a stronger, causal test of this claim would require.

6.2 Why Preparation Reduces Effort

A few things together help explain why preparation cuts down iteration. When context is missing, the AI has to guess what the developer wants, and guesses are often wrong. An AI system, much like a human expert, reasons better when it works from clear constraints and goals instead of an open-ended idea. And more complete context raises the odds that the first attempt lands close to correct, which trims the number of "no, that's not what I meant" exchanges.

This lines up with the objective data in Sections 5.3 and 5.4. High-preparation respondents did not just feel more efficient. They used fewer prompts and fewer re-generations in practice.

6.3 Implications for AI Assistant Design and Industry

For organisations adopting AI-assisted development, these results suggest that training developers on how to prepare before using AI may matter as much as training them on how to prompt. AI assistant builders have a design opening too. Instead of starting from an empty chat box, an assistant could walk a new user through a short, structured set of questions before code generation begins, something like a pre-flight checklist, and flag when important information, such as security requirements, has not yet been given.

6.4 Security, Ethics and Compliance

One part of PDACF that is easy to overlook is its explicit place for security, ethics, and compliance in Phase 4, ahead of implementation. A request such as "build a payment system" gives the AI no direction on encryption, authentication, or audit logging, and the AI is unlikely to add these on its own. Respondents who landed in the best outcome category (A) were also more likely to say they had specified these kinds of constraints upfront. This fits the broader "security by design" principle already established in software engineering [1], [12], [13], and suggests that principle deserves the same attention in AI-assisted workflows that it gets in conventional ones.

6.5 Limitations and Future Work

This study has clear limitations, and its findings should be read with them in mind.

Small sample size (n = 25). The results should be treated as early evidence rather than a definitive conclusion. A larger study, with 100 or more respondents, would support stronger statistical claims.

Limited participant diversity. Most respondents (72%) were final-year students from one university. Results may look different for experienced industry professionals in larger organisations.

Non-standardised tasks. Each respondent worked on a different feature or task, at a different level of complexity, and this study did not control for that.

Single-model perspective. This study did not compare outcomes across different AI models, such as Copilot against ChatGPT against Claude, and some models may well handle vague prompts better than others.

Self-reported measures. Part B of the questionnaire (Q11 to Q15) was designed to check self-reported perceptions against more concrete answers, but the underlying data is still self-reported rather than independently measured, for instance through automated code-quality tools. Self-reported time and prompt counts are also open to recall bias, since respondents estimated these figures shortly after finishing a task rather than logging them as they went.

Correlational design. Every relationship reported in Section 5 is a correlation, not a controlled comparison. Respondents were not randomly assigned to a high- or low-preparation condition; they simply reported their own habits. The findings should be read as a strong association within this sample and should not be generalised beyond it, and they should not be taken as proof that increasing preparation would causally improve outcomes for any individual developer or team.

Future work should take on these limitations directly. A controlled experiment using identical software-development tasks assigned across multiple AI coding assistants (for example, Copilot, ChatGPT, and Claude on the same task) would let preparation level be manipulated rather than only observed, supporting a genuinely causal test. Such a study should also draw on a larger and more diverse participant pool, including more industry professionals across different organisations, and should replace or supplement self-reported outcomes with objective measures such as test-pass rate, static-analysis results, defect density, and independently rated code quality.

7. CONCLUSION

This paper set out the Pre-Development AI Context Framework (PDACF), a six-phase framework, Project Understanding, Requirement Structuring, System Structuring, AI Context Preparation, AI Implementation, and Validation, for preparing software project information before AI-assisted implementation begins. A preliminary survey of 25 respondents found a very strong relationship ($r = 0.940$) between preparation level and reported satisfaction. Respondents who prepared more thoroughly completed tasks 49% faster, used 53% fewer prompts, and needed 65% fewer code re-generations than those who prepared less. Structured preparation before the first AI prompt, then, is not overhead. It is an investment that pays off in time saved and quality gained. This is an early-stage study, and its findings should be taken as a starting point for larger, more rigorous research into how developers, organisations, and AI assistants themselves can better support the preparation stage of AI-assisted software development.

ETHICAL CONSIDERATIONS

Participation in the survey was voluntary, and all respondents gave informed consent before taking part. No personally identifying information was collected; all responses are reported anonymously (R01 to R25). No human subjects were exposed to physical, psychological, or financial risk as part of this study.

CONFLICT OF INTEREST

The authors declare no conflict of interest.

DATA AVAILABILITY

The anonymised survey dataset (Q1 to Q15 responses for all 25 respondents) supporting the findings of this study is available from the corresponding author upon reasonable request.

REFERENCES

1. Bourque, P., & Fairley, R. E. (Eds.). (2014). Guide to the software engineering body of knowledge (SWEBOK), version 3.0. IEEE Computer Society.
2. Sommerville, I. (2015). Software engineering (10th ed.). Pearson.
3. Pressman, R. S., & Maxim, B. R. (2014). Software engineering: A practitioner's approach (8th ed.). McGraw-Hill.
4. Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., ... Amodei, D. (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33, 1877–1901.
5. Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde de Oliveira Pinto, H., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., ... Zaremba, W. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
6. White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., & Schmidt, D. C. (2023). A prompt pattern catalog to enhance prompt engineering with ChatGPT. *arXiv preprint arXiv:2302.11382*.
7. Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., & Zhang, J. M. (2023). Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533*.
8. Peng, S., Kalliamvakou, E., Cihon, P., & Demirer, M. (2023). The impact of AI on developer productivity: Evidence from GitHub Copilot. *arXiv preprint arXiv:2302.06590*.
9. Ronanki, K., Berger, C., & Horkoff, J. (2023). Investigating ChatGPT's potential to assist in requirements elicitation processes. In *Proceedings of the 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. *arXiv:2307.07381*.
10. Norheim, J. J., Rebentisch, E., Xiao, D., Draeger, L., Kerbrat, A., & de Weck, O. L. (2024). Challenges in applying large language models to requirements engineering tasks. *Design Science*, 10, e16.
11. Karpathy, A. (2025, February 2). Vibe coding [Post]. X (formerly Twitter).
12. IEEE Standards Association. (2021). IEEE standard model process for addressing ethical concerns during system design (IEEE 7000-2021). IEEE.
13. Morley, J., Floridi, L., Kinsey, L., & Elhalal, A. (2020). From what to how: An initial review of publicly available AI ethics tools, methods and research to translate principles into practices. *Science and Engineering Ethics*, 26, 2141–2168.
14. Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., & Wang, H. (2024). Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8), 1–79.
15. European Parliament and Council of the European Union. (2024). Regulation (EU) 2024/1689 laying down harmonised rules on artificial intelligence (Artificial Intelligence Act). *Official Journal of the European Union*.