IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 3, May 2025
Impact Factor: 7.67

Developer and LLM Pair Programming: An Empirical Study of Role Dynamics and Prompt-Based Collaboration

Sri Rama Chandra Charan Teja Tadi
Lead Software Developer, Austin, Texas

Abstract: *With the introduction of large language models (LLMs) as coding partners, the classic pair programming dynamics are being rewritten. This research empirically examines the collaboration between software developers and LLMs on software tasks, uncovering a dynamic role toggling informed by prompt accuracy and contextual cues. Instead of deterministic driver-navigator dichotomies, we find an emergent interdependence where programmers function as orchestrators of intent and LLMs oscillate between executor, interpreter, and creative collaborator. Prompt design has emerged as a critical skill for orchestrating collaboration, shifting the focus from code authorship to dialogical problem-solving. This perspective introduces a new vision of human-AI co-creation in coding, highlighting its potential within future intelligent development environments.*

Keywords: Pair Programming, Large Language Models, Human-AI Collaboration, Prompt Engineering, Software Development, Code Co-Creation, Programming Roles, Intelligent IDEs.

I. REFRAMING PAIR PROGRAMMING IN THE ERA OF LLMS

The conventional pair programming model, based on agile software development methodology, is defined by its two positions: the driver and the navigator. The driver actively types code while the navigator inspects every line of code, giving directions, comments, and advice. This model has been conventionally attributed to enhancing the quality of code and collaboration and knowledge sharing among developers [25]. The demand for real-time communication between collaborators calls for increased cognitive investment and, thus, greater learning benefits. With the advent of large language models (LLMs), there is a radical change in this equation that requires redesigning how pair programming functions.

With LLMs like OpenAI's Codex and GPT models becoming increasingly proficient and producing code on their own, integrating them into the paradigm of pair programming demands a conceptual change. This revolution comes with a dismantling of the ancient driver-navigator roles to a more dynamic partnership model. Rather than an individual developer coding and another providing commentary, LLMs can be switch partners that alternate between doing and commenting, essentially doing away with the constraints that defined past roles [11]. This shift encourages developers to perform the role of orchestrators and direct the teamwork process rather than engage in rigid task isolation.

There are several reasons behind investigating this transition. With how quickly AI technology changes, understanding how these updates affect the dynamics of the group, productivity levels, and knowledge acquisition is very important. In addition, existing literature reveals that cooperation with LLMs demands prompt engineering skill, advocating a new set of abilities for programmers [19], [17]. Some of these include effective expression of requests and developing prompts to high-quality code generation, which boosts the interaction between human programmers and AI.

Furthermore, the advent of LLMs introduces industry-scale trends that need to be discovered. Organizations have increasingly employed AI technologies to maximize productivity in software development, creating transformations in how development teams do their work. LLMs have been debated to speed up coding work and eliminate the labor involved in boilerplate programming to allow developers to work on higher-level design and problem-solving [26]. This change demands research on the long-term effects of AI on promoting innovation in software development teams and creating career tracks in the technology sector. Besides, collaboration with developers and LLMs generates new problems and ethical issues. Greater dependence on generative AI, over-reliance on code generated by LLMs, plagiarism issues, and loss of conventional coding abilities are emerging as issues [9], [24]. Empirical studies addressing these problems will give us insights into human-AI collaboration models and the changing software engineering scenario.

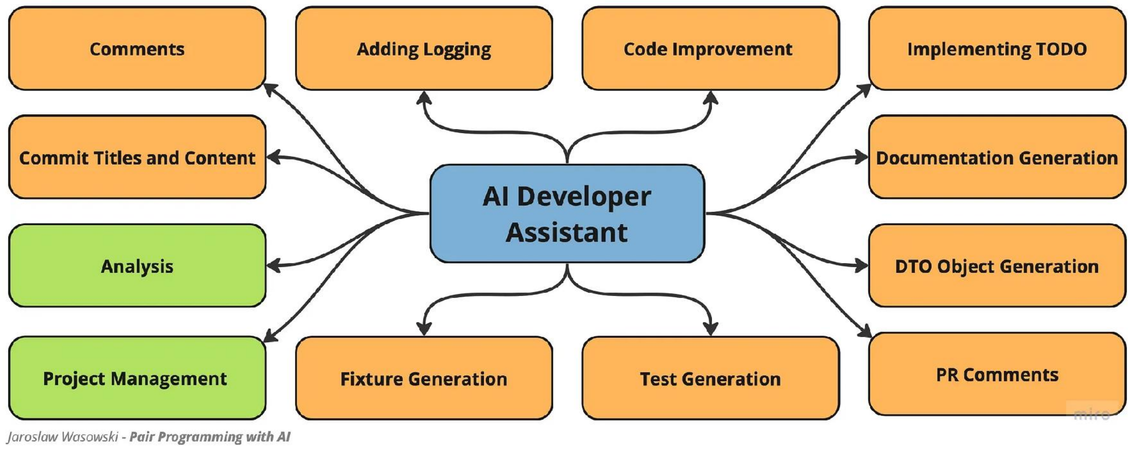


Fig 1: Functional Roles of AI Developer Assistants in Collaborative Software Development
Adapted from [28]

II. OPERATIONALIZING HUMAN-LLM COLLABORATION

In order to successfully explore the human developer-LLM interaction, the empirical research approach used should be well-designed to capture the subtleties of such interaction. The research interest entails a sequence of activities that developers perform, observing how LLMs support or complement their activities in actual coding contexts. Activities involve scratch coding, debugging, and refactoring with the objective of simulating typical software engineering problems encountered in academic environments and actual applications [23]. The activities are chosen to test the limits of LLM performance in making relevant suggestions and producing accurate results.

The subjects of the study are developers of different backgrounds, from novice to advanced programmers, which will give a broad view of how skill levels influence working with LLMs. The subjects are selected from different fields of software development, such as web development, backend programming, and application development, to establish the generalizability of findings in terms of computing [25]. Moreover, the settings where interactions occur, integrated development environments (IDEs), terminal interfaces, and chat interfaces are a significant aspect of the study design since they directly affect the interaction dynamics between LLMs and developers.

Methods of data collection employed include screen recording, interaction logs, and think-aloud protocols, offering a rich and multi-faceted perspective on the collaborative process. Screen captures take in live interaction and decision-making practices, while logs observe individual prompts and LLM output over the long term. The think-aloud protocol takes it further by allowing developers to articulate themselves aloud and shed light on their thought process and utilization of LLM abilities in coding activity [12]. All these capture a rich dataset for robust analysis.

Hence, the research utilizes different analysis methods, such as qualitative coding and behavioral mapping, to examine collaboration behavior. Qualitative coding examines interaction patterns between developers and LLMs, such as dependency, creativity, and conflict. Behavioral mapping enables researchers to map the interaction pattern, highlighting successful moments of collaboration and where LLM performance fell short of developers [10]. This dual strategy offers a complete explanation for the complex dynamics that occur when humans and LLMs collaborate.

III. PROMPT FIDELITY AND CONTEXTUAL GROUNDING

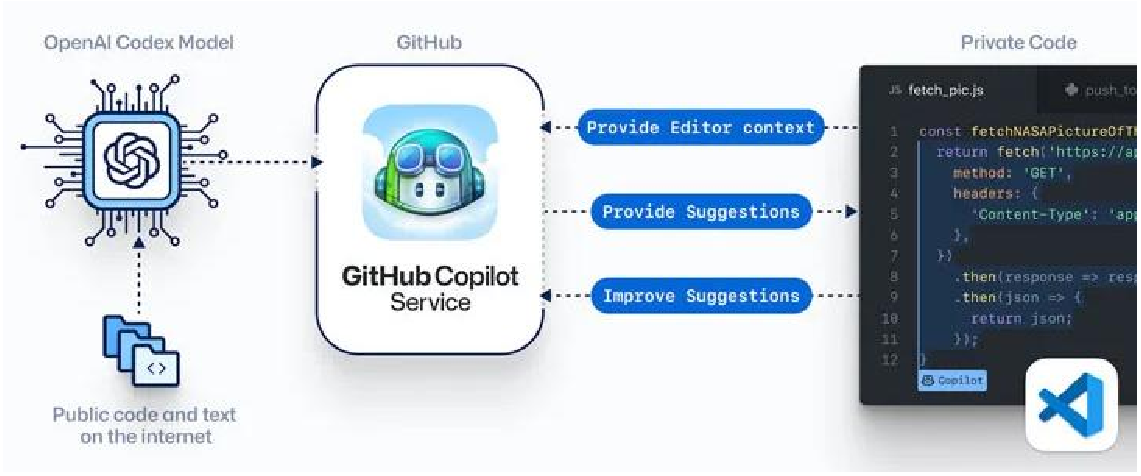
Prompt engineering is vital in setting LLM responses as clarity, structure, verbosity, and order directly impact output accuracy and consistency. For example, well-defined prompts with concise directives enable LLMs to produce semantically adequate and logically sound code, and poorly constructed or excessively wordy prompts have a negative performance cost [6]. Empirical results indicate that directive prompts, with well-defined context and goals, substantially enhance code quality and the transferability of solutions [20]. Literature shows that early fidelity truly plays a role in effective human-AI collaboration via output alignment with intended developer intent [14].

Prompt design structuring is also a determinant of outcome, as revealed by recent empirical findings on chaining sequences of prompts across tasks [22]. It has been noted that the sequencing and layering of commands result in emergent behavior of LLM responses in such a way that previous prompt tokens determine subsequent responses. On the other hand, poorly designed prompts lacking syntactic structure mostly yield fractured or off-topic responses. This contrast emphasizes a strict design procedure based on syntactic and semantic task properties [6].

In addition to structure, session context and memory limitations play an important role in affecting LLM behavior. Although LLMs are sustained during a session, their contextual window can erode, compromising task continuity and semantic anchoring [4]. In extended code construction sessions, say, grounded prompts assist in avoiding possible memory loss by injecting constantly needed context continuously. This does require an iterative process of adapting prompts to the extent that the contextual anchor will remain evocative throughout user interaction so the LLM will be able to maintain high response fidelity [8].

Case studies also illustrate the distinction between ill-defined and well-specified prompts. Those cases where information details are left out usually get responses that have to be further clarified, thus lengthening the development cycle. On the other hand, those prompts with explicit context, well-articulated goals, and structured format get responses that are more accurate and earlier finished subtasks [20], with a greater level of reproducibility and consistency in code generation [14]. This evidence supports the focus on timely fidelity as a core competency in LLM communication.

The tension between prompt engineering and contextual grounding mirrors the dynamic nature of the human-LLM dialogue. It has been demonstrated that iterative and dynamic prompt tuning serves as a correcting mechanism in the case of LLMs going astray [6]. Such interaction not only optimizes individual operations but also preserves semantic coherence as a whole, across sessions, thereby reducing the negative effects of memory capacity [4], [8]. These findings affect contemporary software development and AI-assisted programming methods to date.



```
const fetchNASAPictureOfT
  return fetch('https://ap
    method: 'GET',
    headers: {
      'Content-Type': 'ap
    },
  })
    .then(response => res
    .then(json => {
      return json;
    });
}
```

Fig 2: Architecture of GitHub Copilot: Context-Aware Code Generation Using OpenAI Codex
Adapted from [30]

IV. DYNAMIC ROLE TRANSITIONS IN DEVELOPER-LLM DYADS

Empirical findings show that interaction between LLMs and developers is dynamic rather than static and dynamically changes as a function of task complexity and prompt clarity. In certain observed sessions, the LLM sometimes acts as an explorer, designing novel solutions and highlighting potential shortcomings, while the developer is mainly a verifier

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 3, May 2025
Impact Factor: 7.67

and orchestrator [1]. These role reversals in dynamic form are determined by issues such as uncertainty in prompt instructions and difficulty with the coding task, requiring an adaptive interaction model [2]. Research on the subject indicates that flexibility in roles results in enhanced problem-solving ability and leverages the LLM's creative capabilities [8].

Role identification within these dyads is generally limited to helper, solver, explainer, and verifier. In sophisticated debugging sessions, for instance, LLMs serve as the helper since they present different means of proceeding, while developers serve as the solver in putting and verifying these suggestions [1]. Further, a recurring transition to an explainer function can be noted when LLMs are asked to explain why a coding choice was made, thereby establishing two-way comprehension of the underlying choice-making process [2]. These patterns of observation highlight the adaptive interaction of human reasoning and computational machinery.

The second dynamic role reversal feature is the response behavior mismatch produced by differences in prompt accuracy variations. When confronted with high-fidelity prompts, LLMs perform accurate tasks, and the developers can take a supervisory or verification role [4]. On the other hand, in the case of sudden ambiguity, LLMs can suggest new concepts that force the developers to re-guide the conversation to proceed on course, effectively flipping the conventional roles [8]. Such an exchange illustrates the emergent interdependence of humans and machines in collaborative coding.

Additionally, dynamic role changes are most often a real-time process involving feedback loops. Instructions are adapted according to the quality of LLM output, thus enabling role shifts as the coding environment changes [21]. Refinements through feedback necessitate an ongoing negotiation between computational ability and human expertise, offering a context in which roles are not static but continuously negotiated during the development process. This has been extensively documented in recent empirical studies where role shifts are associated with prompt refinement and task decomposition [2].

A more in-depth analysis of these transformations opens up their effect on overall coding effectiveness and innovation. The resulting roles - from solver to verifier - lend themselves to more robust problem-solving methods and facilitate a co-working that calls upon both human ingenuity and AI-offered hints [1]. Concurrently, variability in the role change also diminishes the risk of overdependence on AI, whereby human oversight always remains at the heart of software development [21], [4]. Such dynamic interactions not only enhance coding quality but also lead to the development of an image of smart development environments.

V. INTENT-ORCHESTRATION AS A DEVELOPER COMPETENCY

Intent orchestration in LLM-augmented coding is the method where developers transition from coding to that of conductors of semantic intent. In the new model, developers set high-level goals and iteratively adjust prompts to guide LLM output toward a solution that has been integrated. This is an art that extends beyond the mechanics of code-making and relies on knowledge of how language models understand subtlety and context. It has been found that the skill of designing specific prompts is the foundation of this emergent skill set and enables developers to dynamically specify and tune semantic targets [19].

Developers become deliberate orchestrators by iteratively refining prompts and hypothesis testing by engaging in conversation with the LLM. Different patterns of prompts are initially tried out, and more and more, it is discovered which patterns produce the best and contextually suitable outputs. This circular process has been termed as essential to ensure LLM responses are aligned with developers' overall objectives [11]. Practically speaking, this now means that developers are no longer concerned with the syntax of code but are seeking to perfect natural language conversation-driving code generation. Iterative approaches have also been proven to improve transparency and eventually make co-generated code more reliable [17].

Empirical observation of study data has highlighted this vocational shift. The developers have indicated, in interviews and audited sessions, how confidence is built up step by step to cope with convergent outputs through paraphrasing, exposition, or downscaling their cues. Such interaction is hypothesis testing through iterative prompt adjustment, where competing coding solutions are tried out and task semantics bargained with the LLM [12]. In addition, successful examples prove that well-coordinated intent can help counter ambiguities in LLM outputs, with the resulting final output following the original design intent very closely.

New skills here involve the ability to iterate prompts with accuracy, carry out hypothesis testing through natural language conversation, and control divergent outputs in real time. A methodological strategy to cause engineering not only improves the uniformity of output but also sharpens the ability to deal with emergent model behavior [16]. In addition, dealing with varying outputs necessitates a resilient mindset, where every iterative correction is regarded as a mini-experiment, a process referred to as key to programming practice in the age of generative AI [9]. Together, these capabilities constitute the basis of intent orchestration, broadening the set of abilities involved in software development today.

As a larger implication, the shift toward intent orchestration as a fundamental developer skill has large-scale implications for future software engineering practices. As developers take on an increased role as orchestrators, they fuel a transition from code authorship to strategic leadership, shaping software system structure and design. All such changes have proven to improve effectiveness in overall development and open prospects for innovative solutions to problems [27]. Furthermore, the ability of such competencies to fundamentally change work and transform it has been highlighted where technical competencies are augmented with high-order communications and analytical capacities [26]. In this context, the emphasis on intent orchestration signals a broader redefinition of software development proficiency in the age of human-AI collaboration.

VI. DIALOGUE-DRIVEN PROBLEM SOLVING VS. CODE AUTHORSHIP

The availability of LLM-based problem-solving is a stark break from the conventional linear programming approach. Conventional approaches generally were plan-code-debug, and coding was the obvious highlight of a programmer's technical abilities. The new methods, however, solve intricate issues by way of back-and-forth communication in terms of natural language haggling and auto-coding. It has been proven that although LLMs are capable of performing more extensive programming tasks, conversation stimulates ideation, and hence, communication becomes a core problem-solving mechanism [3]. This paradigm changes the perspective from static code generation to dynamic, conversational interaction.

Iterative decomposition of the problem now takes place through a Q&A of descriptive questions and answers, allowing developers to decompose high-level tasks into manageable subtasks interactively with the LLM. The Q&A process enables the LLM to ask descriptive questions that encourage developers to elaborate on their initial descriptions of the problem. This Q&A process has been found to reveal latent requirements and edge-case issues, allowing for a more comprehensive exploration of the problem space [1]. By repeating specific questions and responses, developers can then capture nuances that would otherwise be lost in a typical coding pipeline.

Examples of some of these exchanges include when developers have explanatory discussions with the LLM to test understanding or iteratively refine a design strategy. There have been instances in which the LLM poses follow-up questions that prompt developers to reconsider boundary conditions and restate tasks [24]. This process, through conversation, generates solutions co-created as a result of the sharing of ideas, the LLM clarifying and also an ideation partner. The natural language negotiation inherent in conversation-based problem-solving has also been found to not only offer more clarity to the process but also allow for more mutual understanding of the task itself [19].

The influence of this dialogue process on developer cognition, process, and ownership is significant. Developers indicate that conversational problem-solving leads to high levels of cognitive flow, with repeated conversation resulting in accelerated refinement and creativity. These types of interactions have been said to create a sense of uninterrupted co-creation, where human wisdom and LLM ability collaborate [4]. This interactive collaboration unburdens the cognitive labor historically left to debugging or code creation by distributing the work across machines and humans. As a result, greater creativity and re-established ownership of the entire design process are typically claimed.

In essence, conversational problem-solving redefines the developer's role as that of a facilitator in an advanced conversation with AI. This new model places iterative negotiation and clarifying conversation above code production alone, and it is now believed that successful problem-solving relies as much on communication quality as on technical ability. This new line of thinking has been claimed to make developers think more exploratorily, to value idea iteration

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 5, Issue 3, May 2025 Impact Factor: 7.67

and polishing in collaboration [17]. Furthermore, this change results in a future where developers are no longer simply programmers but effective negotiators as part of a human-AI collaborative creative process [26].

VII. DESIGN IMPLICATIONS FOR INTELLIGENT DEVELOPMENT ENVIRONMENTS

This research's empirical findings highlight the need to craft development environments into intelligent, interactive ones that foster the subtleties of human-LLM partnerships. The editors of code should not merely be future IDEs but environments that facilitate iteratively experimenting prompts, role adaptation, and heavy-duty context maintenance. In practice, this involves rolling out features like prompt scaffolding modules that nudge developers to craft and improve their questions in a productive and efficient manner and inform them about prompt clarity and organization in real time. This scaffolding can be in the form of templates and context-dependent cues based on prior interactions, similar to adaptive communication in agile techniques [21]. By incorporating a visual layer that translates conversation flow into code modification, such IDEs can provide more accurate semantic anchoring so that the output of the LLM remains aligned with the intention of the developer.

Making prompt scaffolding features native to the IDE is perhaps the most important recommendation. This would be an interactive sidebar pane providing suggestive prompt structure and setting recommendations, utilizing models such as those employed in recent works, which actually reduce errors during LLM conversations [8]. Designers would welcome having an engine that not only maintains syntactical correctness but also points out semantic context deficiencies and thus serves as a "natural language command prompt linter". Such a support would be based on iterative refinement cycles to accelerate communication and alleviate the friction of constantly switching between routine coding tasks and natural language negotiation [6]. Interleaving these proposals with performance history allows developers to know which prompt patterns optimize outcomes.

Beyond prompt scaffolding, the visualization of context in smart IDEs is very significant in dealing with the generally large-scale conversation between developers and LLMs. More sophisticated visualizations may involve annotated timelines with markers for changes in function, whether the LLM is serving as assistant, solver, or creative partner, and an ongoing mapping of history and decision-making sensitive to context [22]. Such visualization would enable developers to keep tabs on session history and not get lost in changing contexts due to memory constraints caused by LLM architectures. Merging such graphical aids with built-in debugging and code inspection capabilities can result in an efficient cycle of feedback, providing insight into computational thinking alongside hints where the conversation deviated from the initial focus. Such capabilities have already begun to emerge in commercial tools like GitHub Copilot but are available for more subtle and dynamic applications [5].

Smart development environments must include role-awareness hints and trust tokens as well to make role switching easy. For example, the IDE can offer dynamic role labels, e.g., "ideation mode," "code execution mode," or "debug mode", to remind programmers of the active role of the LLM and the human developer. This level of transparency can improve trust by providing reliability measures or confidence ratings for generated code fragments, thereby facilitating an empowered decision-making process [21]. Trust indicators could also be connected to code provenance information, providing developers with insight into the source and purpose of LLM suggestions. These characteristics would enable developers to switch easily between control and creative collaboration, bringing more stability and consistency into the development process.

To make continuous collaboration possible, there is a need to ensure upcoming IDEs provide continuous, iterative feedback loops for the sake of timely experimentation and context preservation. An option for "history replay" could permit developers to revisit previous iterations, compare prompt evolution, and comprehend how slight changes impacted results. This reflective ability would not only enable ongoing learning but also enhance long-term trust in LLM output by providing transparent derivations of code suggestions. In addition, the IDE must be able to integrate with version control tools and code repositories so that incremental updates are stored systematically and readily available. In doing so, smart development environments become not only a tool of immediate productivity but also a long-term investment in developer ability and code quality through documented collaboration [20], [15].

VIII. CHALLENGES, LIMITATIONS, AND ETHICAL FRICTIONS

Despite the optimistic advances provided by LLM-aided programming, there are some inherent limits and difficulties that need to be recognized. One of the most apparent limits is the variation in LLM output due to inherent model inclinations and sampling randomness. With a limited sample and task-specific restrictions in empirical experiments, variation produces doubt about the generalizability of research results to larger programming contexts [13]. In addition, although iterative prompt engineering has brought substantial benefits, it has also been reported that prompt quality is still very variable and can lead to unreliable, irreproducible outcomes. This uncertainty puts the stability of co-created codebases to the test and heightens the value of having robust, systematic methods for future studies. Moreover, technical limitations, including constrained memory windows in LLMs, jeopardize sustained context retention in prolonged interactions, indicating that the sophistication of human-LLM collaboration is not yet fully manageable.

Over-reliance on LLM proposals is another authentic concern. Programmers may be suffering from cognitive offloading, with machine-generated proposals being relied upon too much, gradually losing the subtle understanding required for productive debugging and optimization of codes. This reliance can make the human coder less invested in the underlying code, potentially leading to lower skill retention and lower code explainability [15]. In addition, the tendency to blindly accept LLM outputs without adequate critical review can further worsen the propagation of errors or even inject security vulnerabilities into codebases. Such risks necessitate the use of mechanisms that enforce active human supervision and cause constant auditing of code produced by AI. Establishing trustworthy proxies for trust and constant feedback loops is essential in avoiding the loss of essential programming ability.

Ethical dilemmas also arise in the publication of LLM-aided tools. One prominent concern is authorship confusion, with the difference between human-created content and AI-suggested input being eroded. This confusion raises concerns about intellectual property rights, licensing of code, and attribution, which are of greatest importance in open-source and proprietary software environments [7]. There is also the risk that LLMs can propagate biased or insecure coding practices unwittingly, particularly if models are trained from data sets imbued with hidden historical biases or weaknesses. These concerns are already bringing forth demands for more regulatory control and stricter ethical guidelines to oversee the use of LLMs in software development. The spreading of biased code both weakens software security and entails a sacrifice to the industry's ethical foundation, which renders a proactive approach to detecting and preventing bias a necessity.

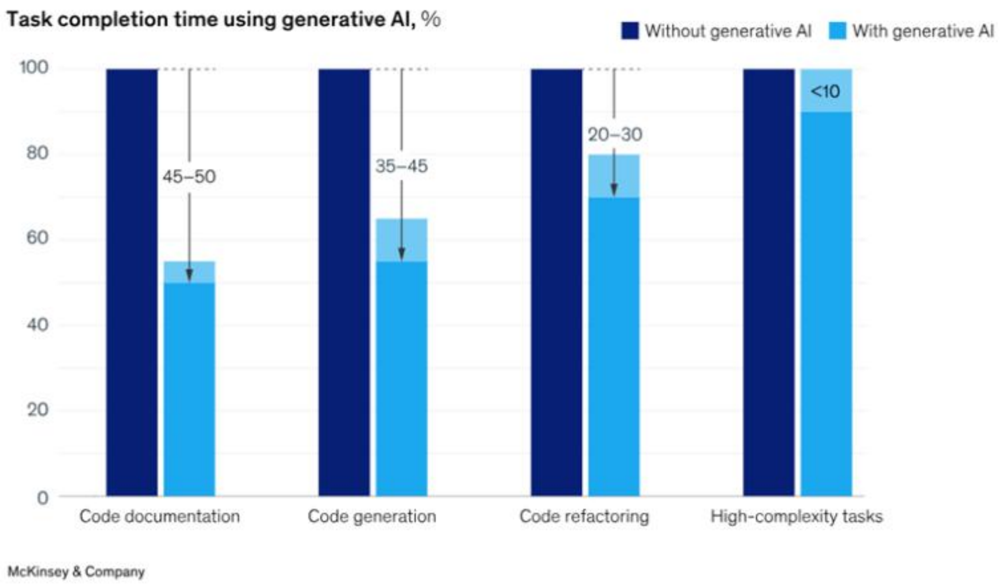


Fig 3: Generative AI can increase developer speed, but less so for complex tasks
Adapted from [29]

The design of the study has limitations of its own that should be given serious consideration. Methodologically, task choice and sample size both necessarily place boundaries on experiment generalizability such that results are difficult to extrapolate into wide varieties of real-world programming environments [18]. Furthermore, dynamic role-shifting and variation based on prompts might be unsalvageable in an artificial lab experimental design. These experimental limitations highlight the necessity for follow-up studies with larger and more heterogeneous participants and tasks cutting across a wide range of software development issues. To address them, new data acquisition techniques and sophisticated types of analysis must be employed to attain a better understanding of human-LLM collaboration dynamics and their implications.

IX. CONCLUSION

In conclusion, while LLM-assisted programming unlocks new possibilities for computer programming, it is also filled with fundamental challenges and ethical concerns. There would be a requirement for future investigations into creating universally acceptable guidelines on prompt engineering, setting best practices for tracking and fixing biases, and defining hard-and-fast rules for authorship and licensing code in work created by AI. Further, measures like heightened transparency mechanisms, real-time feedback loops, and proactive human monitoring procedures need to be incorporated in order to ensure ethical and responsible LLM technology deployment. As these challenges are addressed through further empirical research and the integration of ethical design principles, the software engineering community will be better positioned to harness the full potential of human-LLM collaboration while safeguarding the technical robustness and ethical standards of modern development practices.

REFERENCES

[1] Zhang, H., Cheng, W., Wu, Y., and Hu, W., "A pair programming framework for code generation via multi-plan exploration and feedback-driven refinement," 2024, pp. 1319–1331. doi: https://doi.org/10.1145/3691620.3695506

[2] Khurana, A., Subramonyam, H., and Chilana, P., "Why and when LLM-based assistants can go wrong: Investigating the effectiveness of prompt-based interactions for software help-seeking," 2024, pp. 288–303. doi: https://doi.org/10.1145/3640543.3645200

[3] Hou, W. and Ji, Z., "Comparing large language models and human programmers for generating programming code," *Advanced Science*, vol. 12, no. 8, 2024. doi: https://doi.org/10.1002/advs.202412279

[4] Loya, M., Sinha, D., and Futrell, R., "Exploring the sensitivity of LLMs' decision-making capabilities: Insights from prompt variations and hyperparameters," 2023, pp. 3711–3716. doi: https://doi.org/10.18653/v1/2023.findings-emnlp.241

[5] Wong, M., Guo, S., Hang, C., Ho, S., and Tan, C., "Natural language generation and understanding of big code for AI-assisted programming: A review," *Entropy*, vol. 25, no. 6, p. 888, 2023. doi: https://doi.org/10.3390/e25060888

[6] Luo, Y., Tang, Y., Shen, C., Zhou, Z., and Dong, B., "Prompt engineering through the lens of optimal control," *Journal of Machine Learning*, vol. 2, no. 4, pp. 241–258, 2023. doi: https://doi.org/10.4208/jml.231023

[7] Choquet, G., Aizier, A., and Bernollin, G., "Exploiting privacy vulnerabilities in open-source LLMs using maliciously crafted prompts," 2024. doi: https://doi.org/10.21203/rs.3.rs-4584723/v1

[8] Singh, S., Guleria, A., and Shukla, V., "Unlocking AI-powered conversations and code excellence: Exploring prompt patterns in conversational AI and software engineering," *International Research Journal of Modernization in Engineering Technology and Science*, 2024. doi: https://doi.org/10.56726/irjmets47950

[9] Denny, P., Leinonen, J., Prather, J., Luxton-Reilly, A., Amarouche, T., Becker, B., et al., "Prompt problems: A new programming exercise for the generative AI era," 2024, pp. 296–302. doi: https://doi.org/10.1145/3626252.3630909

[10] Kim, J., Lee, S., Han, S., Park, S., Lee, J., Jeong, K., et al., "Which is better? Exploring prompting strategy for LLM-based metrics," 2023. doi: https://doi.org/10.18653/v1/2023.eval4nlp-1.14

[11] Uzair, W. and Naz, S., "Six-tier architecture for AI-generated software development: A large language models approach," 2023. doi: https://doi.org/10.21203/rs.3.rs-3086026/v1

[12] Joseph, T. and Keneth, M., “Exploring the synergy of grammar-aware prompt engineering and formal methods for mitigating hallucinations in LLMs,” *East African Journal of Information Technology*, vol. 7, no. 1, pp. 188–201, 2024. doi: https://doi.org/10.37284/eajit.7.1.2111
[13] Tosi, D., “Studying the quality of source code generated by different AI generative engines: An empirical evaluation,” *Future Internet*, vol. 16, no. 6, p. 188, 2024. doi: https://doi.org/10.3390/fi16060188
[14] Santu, S. and Feng, D., “TELER: A general taxonomy of LLM prompts for benchmarking complex tasks,” 2023. doi: https://doi.org/10.18653/v1/2023.findings-emnlp.946
[15] Thistleton, E. and Rand, J., “Investigating deceptive fairness attacks on large language models via prompt engineering,” 2024. doi: https://doi.org/10.21203/rs.3.rs-4655567/v1
[16] Li, W., Chen, X., Deng, X., Wen, H., You, M., Liu, W., et al., “Prompt engineering in consistency and reliability with the evidence-based guideline for LLMs,” *NPJ Digital Medicine*, vol. 7, no. 1, 2024. doi: https://doi.org/10.1038/s41746-024-01029-4
[17] Velásquez, J., Franco, C., and Cadavid, L., “Prompt engineering: A methodology for optimizing interactions with AI-language models in the field of engineering,” *Dyna*, vol. 90, no. 230, pp. 9–17, 2023. doi: https://doi.org/10.15446/dyna.v90n230.111700
[18] Weisz, J., Müller, M., Houde, S., Richards, J., Ross, S., Martinez, F., et al., “Perfection not required? Human-AI partnerships in code translation,” 2021, pp. 402–412. doi: https://doi.org/10.1145/3397481.3450656
[19] Fagbohun, O., Harrison, R., and Dereventsov, A., “An empirical categorization of prompting techniques for large language models: A practitioner’s guide,” *JAIMLD*, vol. 1, no. 4, pp. 1–11, 2023. doi: https://doi.org/10.51219/jaimld/oluwole-fagbohun/15
[20] Aljanabi, M., Yaseen, M., Ali, A., and Mohammed, M., “Prompt engineering: Guiding the way to effective large language models,” *Iraqi Journal for Computer Science and Mathematics*, pp. 151–155, 2023. doi: https://doi.org/10.52866/ijcsm.2023.04.04.012
[21] Priyadharasini, M., Sriram, S., T, S., and Vigneshwaran, N., “Steve Jobs: Pioneering AI in software engineering,” *International Research Journal of Advanced Engineering Hub*, vol. 2, no. 4, pp. 823–829, 2024. doi: https://doi.org/10.47392/irjaeh.2024.0116
[22] Wu, T., Terry, M., and Cai, C., “AI chains: Transparent and controllable human-AI interaction by chaining large language model prompts,” 2021. doi: https://doi.org/10.48550/arxiv.2110.01691
[23] Kokol, P., “The use of AI in software engineering: A synthetic knowledge synthesis of the recent research literature,” *Information*, vol. 15, no. 6, p. 354, 2024. doi: https://doi.org/10.3390/info15060354
[24] Sun, J., Liao, Q., Müller, M., Agarwal, M., Houde, S., Talamadupula, K., et al., “Investigating explainability of generative AI for code through scenario-based design,” 2022, pp. 212–228. doi: https://doi.org/10.1145/3490099.3511119
[25] Barenkamp, M., Rebstadt, J., and Thomas, O., “Applications of AI in classical software engineering,” *AI Perspectives*, vol. 2, no. 1, 2020. doi: https://doi.org/10.1186/s42467-020-00005-4
[26] Özkaya, İ., “Application of large language models to software engineering tasks: Opportunities, risks, and implications,” *IEEE Software*, vol. 40, no. 3, pp. 4–8, 2023. doi: https://doi.org/10.1109/ms.2023.3248401
[27] Marar, H., “Advancements in software engineering using AI,” *Computer Software and Media Applications*, vol. 6, no. 1, p. 3906, 2024. doi: https://doi.org/10.24294/csma.v6i1.3906
[28] J. Wasowski, “Pair Programming with AI - Principal & Junior Developer Support,” *Medium*, 2023. [Online]. Available: https://medium.com/@wasowski.jarek/pair-programming-with-ai-code-tests-comments-documentation-f1c26d342af0
[29] D. Keller, “5 Ways AI Pair Programmers Impact Developer Productivity,” *Five Blog*, 2023. [Online]. Available: https://five.co/blog/ai-pair-programmers-impact-on-developer-productivity/
[30] Haqtify, “GitHub Launches ‘Copilot’ — AI-Powered Code Completion Tool,” *Haqtify News*, 2021. [Online]. Available: https://haqtify.com/news/ai/github-launches-copilot-ai-powered-code-completion-tool/