Trust Dynamics in AI-Assisted Development: Definitions, Factors, and Implications

Sadra Sabouri[♠], Philipp Eibl[♠], Xinyi Zhou[♠], Morteza Ziyadi[◇], Nenad Medvidovic[♠], Lars Lindemann[♠] and Souti Chattopadhyay[♠]

[♠]Department of Computer Science, University of Southern California, Los Angeles, California
[◇]Amazon AGI
[♠]{sabourih, eibl, xzhou141, neno, llindema, schattop}@usc.edu, [◇]mziyadi@amazon.com

***Abstract*—Software developers increasingly rely on AI code generation utilities. To ensure that "good" code is accepted into the code base and "bad" code is rejected, developers must know when to trust an AI suggestion. Understanding how developers build this intuition is crucial to enhancing developer-AI collaborative programming. In this paper, we seek to understand how developers (1) define and (2) evaluate the trustworthiness of a code suggestion and (3) how trust evolves when using AI code assistants. To answer these questions, we conducted a mixed-method study consisting of an in-depth exploratory survey with (n=29) developers followed by an observation study (n=10).**

We found that comprehensibility and perceived correctness were the most frequently used factors to evaluate code suggestion trustworthiness. However, the gap in developers' definition and evaluation of trust points to a lack of support for evaluating trustworthy code in real-time. We also found that developers often alter their trust decisions, keeping only 52% of original suggestions. Based on these findings, we extracted four guidelines to enhance developer-AI interactions. We validated the guidelines through a survey with (n=7) domain experts and survey members (n=8). We discuss the validated guidelines, how to apply them, and tools to help adopt them.

***Index Terms*—Trust, AI-code assistants, Software development**

I. Introduction

The usage of LLM-based code assistants is becoming increasingly common among developers. These AI-code assistants, like GitHub Copilot, PaLM 2 [1], and ChatGPT 4 [2], have garnered massive popularity with reports of over a million developers using Copilot as of late 2023. More recent reports from 2024 suggest that 76% of developers are using or plan to use AI tools, and that 72% of developers are favorable or very favorable towards AI tools for development [3].

Using LLMs fundamentally changes the cognitive process underlying the process of programming. Compared to traditional problem-solving approaches of programming without AI, where programmers ideated solutions for a problem, developers now increasingly refer to AI to identify solution approaches. This shifts the role of developers from generating code to understanding, selecting, and adapting code.

Under such a paradigm, developers must carefully decide which suggested solutions to trust and integrate. It is untrivial to make these decisions. Recent studies have indicated that issues with user prompts often lead to incorrect or suboptimal code suggestions [4]. Code suggestions can have security [5] and vulnerability issues [5], and robustness [6]. Additionally, decisions to trust code suggestions are not independent of subsequent decisions. Accepting a bad code suggestion can lead to wasted time fixing bugs, resolving conflicts, and undoing changes.

Yet, developers lack support in deciding which suggestion to trust in real time. One body of work quantified and calculated different model parameters like a "confidence score" to convey trustworthiness [7], [8]. However, these scores were not meaningful, as they became evident when tools like Copilot removed such features from the interface [5]. Others tried attempts to adopt the LLM architecture to increase trustworthiness [9], [10], which were also found to be ineffective [11].

The lack of support for developers stems from our lack of understanding of what developers consider a trustworthy code suggestion. The importance of supporting the underlying process is imminent from the increasing number of very recent publications looking into the dynamics of what influences developer trust [12], [13]. Without this knowledge, we will not be able to design meaningful support that improves developers' ability to decide which code suggestions to accept and, eventually, how to use AI code assistants better.

To analyze the developers' decisions in estimating trustworthiness of code suggestions, we investigate ***RQ1***. *How do software developers define a trustworthy code suggestion?* The definition tells us about the support developers need to make better decisions. Next, we investigate how developers evaluate AI-suggested code in practice by asking ***RQ2****: What factors do developers use when assessing trustworthiness?* Any gap between definition and evaluation in practice points to a gap in support for developers. The lack of such support might influence developers to reconsider previously trusted code. This leads us to ask ***RQ3****: Why do developers alter their trust in code suggestions?* Understanding the patterns behind these changes can help developers avoid unnecessary effort and make better trust-related decisions.

We conducted a mixed-method study with an exploratory survey with (n=29) developers to delve deeper into how developers make decisions about the trustworthiness of code suggestions. In parallel, we conducted an observation study with a short post-study interview session with (n=10) developers and student programmers, providing triangulating findings.

Our findings suggest that while trustworthy code is traditionally regarded as reliable and secure, developers using AI assistants prioritize correctness, comprehensibility, and maintainability to estimate trustworthiness. However, there is a lack of support for evaluating these characteristics on the fly, leading developers to assess proxy characteristics instead. Notably, developers mistakenly trust code suggestions in 48% of observed instances (142 in total), often due to incorrect assessment of correctness or blindly accepting code.

Based on our findings, we provide four guidelines for evaluating the trustworthiness of code suggestions and pathways to apply and build tools to help apply the guidelines. Our paper makes a novel effort to delve deep and look into how developers define, assess, and build ideas about trustworthy suggestions from AI code assistants. Findings from our study can inform future code-AI tools or frameworks as they provide a meaningful data-driven understanding of the cognitive process of human-AI collaborative programming.

II. Related Work

In this paper, we aim to study the dynamics of how developers decide which code suggestions to *trust* and how they define/assess the *trustworthiness* of code suggestions. Throughout this paper, we use *trust* to refer to the action of developers relying on/accepting a generated code suggestion and *trustworthiness* as the quality of a code suggestion that indicates the degree to which developers can trust it. Our paper builds on two bodies of work:

A. Generative AI in software development

Advances in generative AI models have opened up countless new possibilities in software development [14], [15], and AI-code assistants like Copilot and PaLM 2 [1] have gained popularity in their ability to generate meaningful suggestions. Nonetheless, generating production-quality code suggestions for complex software systems remains a difficult task, especially in fast-paced and stressful work environments [16], [17].

AI code generation tools shift the cognitive process of programming [18], [19], introducing challenges like incomplete mental models [20] and difficulty in evaluating AI-suggested code [14], [15], [21]. These barriers make it harder for developers to decide which suggestions to trust [5], [22], especially given the uncertainty and lack of accountability in AI outputs. These limitations highlight the need to carefully evaluate suggested code and align with recent frameworks for building trustworthy software tools [23]. This motivates our study of trust mechanisms developers use when working with AI code assistants.

B. Trustworthy software

Bohem et al. introduced a hierarchical categorization of code characteristics. At the foundational level, they identified characteristics such as accuracy, robustness, and consistency. These foundational traits were then grouped into broader categories: reliability, efficiency, testability, understandability, modifiability, and human engineering [24]. Since then, software researchers have used quantitative metrics to assess different characteristics of the code [25] such as comprehensibility [23], [26], readability [27], correctness, usability, efficiency, and maintainability. However, recent reports highlight the challenges of relevancy [28] and empirical support [29] for using software metrics in estimating code characteristics.

Complex characteristics like the trustworthiness of code suggestions that depend on individual developer expectations and requirements are especially different from software metrics. Investigating trust requires an understanding of human-centric metrics around code interactions [30]. To enhance the trustworthiness of AI models, some have proposed selectively displaying results to users based on specific code characteristics [31]. While this approach may improve developer-AI interactions, it doesn't tackle the fundamental issue of building trust in the AI system. Inspired by the potential of using concepts of code characteristics, we evaluate the effectiveness of code factors in understanding how developers evaluate the trustworthiness of code suggestions.

III. Method

A. Survey

We conducted a survey in two parts. In the first part, we asked participants to share their definitions of trust in software development and identify factors influencing their trust. Then, we asked them to rate Python code snippets based on how trustworthy they think that code snippet is.

Participants. We recruited 29 participants, including 14 professionals (~50%), by email or direct message on professional social media, such as LinkedIn. Participants were compensated $15 via an Amazon gift card as an incentive for participating. The only criteria used for recruitment were that the participants were older than 18 years, residents of the United States, and familiar with Python.

Demographics. We asked for participants' age, gender identity, highest level of completed education, current job title, years of programming experience, and years of Python programming experience. They were 22 to 39 years old. 17 were male, 11 female, and one preferred not to identify their gender. There were ten participants with a bachelor's as their highest completed degree, 18 with a master's degree, and one with a high school diploma or equivalent. There were 15 graduate students, 13 software engineers and data scientists, and one aerodynamics engineer. Their years of experience in software engineering ranged from less than a year to ten years (μ = 4.66, σ = 3.24). Similarly, their Python programming language experience ranged from none to ten years. Details of participants are provided in Table I. Out of 29 participants, 22 had prior experience with AI code assistants. Half of those 22 said they *prompt* the AI code assistant for code suggestions occasionally (25% of the time). 13 said they *accept* the models' suggestion sometimes (50% of the time).

Factor Ratings. We asked participants to rate factor's contribution to forming their trust in a code suggestion. We started with the software metrics collected from software quality assessment packages such as SonarQube [32], Radon

TABLE I
SURVEY PARTICIPANTS DEMOGRAPHICS

	Age	Gnd.	Job	Exp.	I	II
P1	25	M	Student	0	Rarely (0%)	Rarely (0%)
P2	25	M	Student	3	Occasionally (25%)	Sometimes (50%)
P3	24	M	Student	3	Rarely (0%)	Occasionally (25%)
P4	28	M	Software Developer	3	Frequently (75%)	Occasionally (25%)
P5	24	F	Student	2	Occasionally (25%)	Sometimes (50%)
P6	30	M	Student	10	Sometimes (50%)	Sometimes (50%)
P7	30	M	Software Developer	10	Rarely (0%)	Sometimes (50%)
P8	24	M	Student	-	-	-
P9	27	F	Student	3	Sometimes (50%)	Rarely (0%)
P10	24	M	Student	1	Sometimes (50%)	Rarely (0%)
P11	27	F	Student	1	-	-
P12	27	F	Software Developer	6	-	-
P13	23	M	Student	1	-	-
P14	25	M	Data Scientist	7	Occasionally (25%)	Sometimes (50%)
P15	28	F	Software Developer	5	-	-
P16	27	F	Software Developer	6	Occasionally (25%)	Occasionally (25%)
P17	24	M	Student	0	-	-
P18	36	F	Student	6	Frequently (75%)	Sometimes (50%)
P19	-	-	Student	5	Occasionally (25%)	Rarely (0%)
P20	26	F	Student	4	Frequently (75%)	Sometimes (50%)
P21	24	F	Student	6	Occasionally (25%)	Sometimes (50%)
P22	33	F	Software Developer	9	Occasionally (25%)	Frequently (75%)
P23	28	M	Software Developer	5	Occasionally (25%)	Sometimes (50%)
P24	33	M	Software Developer	10	Sometimes (50%)	Sometimes (50%)
P25	28	F	Software Developer	3	Occasionally (25%)	Sometimes (50%)
P26	39	M	Software Developer	4	-	-
P27	30	M	Software Developer	9	Sometimes (50%)	Sometimes (50%)
P28	-	M	Software Developer	10	Sometimes (50%)	Rarely (0%)
P29	22	M	Software Developer	3	Sometimes (50%)	Sometimes (50%)

*I: How often do you prompt AI while programming?
*II: How often do you accept AI suggestions while programming?

[33], Bandit [34], and Codacy [35]. Two researchers clustered those metrics into seven categories. These categories were 1) *Size*, 2) *Maintainability*, 3) *Comprehensibility*, 4) *Similarity to codebase*, 5) *Correctness*, 6) *Optimality*, and 7) *Security*. We asked participants to rate each factor on a scale from 1 (not affecting trustworthiness) to 5 (affecting trustworthiness a lot).

"Trust" Definition. We also asked participants for their definition of a trustworthy code suggestion by asking "*What is a trustworthy code suggestion for you? In other words, when do you trust a code suggestion to accept it (into the codebase)?*" One author open-coded the trust definition for each participant and performed a round of negotiated agreements with a second author to identify 13 dimensions in participants' trust definition and presented in RQ1.

In the second part, we gather participants' trust ratings on various code snippets and comments to understand why they trust a snippet.

Code Snippet Dataset. We used code snippets from the *CodeXGlue* dataset [36] for its relevance and availability. It provides code-to-text pairs in multiple programming languages. Other datasets we considered, such as DiverseVul [37] and CVEfixes [38], were either limited to less popular languages or had usability issues. The dataset only contained Python code snippets. We scoped our study to Python for its simplicity [39], popularity among scientists [40], and extensive supportive community [39] to broaden our participant range.

Of the 251,820 Python functions in the *CodeXGlue* dataset, we filtered out functions longer than 20 lines and shorter than 10 lines—following prior work [41]—narrowing the set to 96,369 functions. This decision helped ensure that the functions are not too time-consuming to parse or too short to evaluate. We then randomly sampled 100 snippets, which two authors categorized into "Easy", "Medium", and "Hard" based on understandability. Using inter-rater agreement to ensure consistency in the categorization, we reached kappa 0.62, signifying substantial agreement according to Cohen's kappa coefficient [42] after two rounds (round one 0.03 with $p = 0.018$ and round two: 0.62 with $p = 0.037$). We prompted GitHub Copilot with function signatures and docstrings to generate AI versions of each snippet. The authors then manually inspected these 100 generated snippets to ensure they were meaningful. Since all snippets were derived from reviewed GitHub code, they can be considered correct.

Code Trust Evaluation. We randomly selected 20 code snippets for each participant, asking them to rate the trustworthiness of each snippet on a scale from 1 (least trusted) to 10 (most trusted). To provide context to evaluate the snippets, we included a brief one-line project description for each snippet, avoiding using full code bases or links to GitHub to prevent bias. The selected snippets were evenly split between human-generated and AI-generated code and were categorized by difficulty into "Easy", "Medium", and "Hard".

We asked participants to comment on their trust ratings by answering "*What were your reasons to (not to) trust a code snippet?*" Comment boxes were placed on every three medium-difficulty snippets and on every two hard and easy-difficulty snippets to reduce fatigue, given the higher number of medium snippets. This approach gave us a dataset of annotated and commented code snippets. Two researchers conducted five rounds of inter-rater reliability assessment for hybrid qualitative coding, starting with an initial codebook based on *"Trust" Definition* codes [30], and [43]. The kappa values for inter-rater agreement were 0.0136 ($p = 0.8$), 0.426 ($p < 0.001$), 0.496 ($p < 0.001$), 0.574 ($p < 0.001$), and 0.697 ($p < 0.001$). After revising the codebook and merging codes, they achieved a strong agreement [42] with a kappa of 0.844 ($p < 0.001$) in the final round.

B. Observational Study

We conducted an observation study with a post-study interview to identify subconscious patterns in how people evaluate the trustworthiness of code. This study provided the observational study data and post-session interview data sources from which we can answer RQ3. We used Qualtrics for the pre-study questionnaire and Zoom for the interviews, both approved by the Institutional Review Board (IRB).

Participants. We recruited ten participants (five professional developers and five student developers) via university email lists and professional social media. Recruitment was balanced (a student and a software developer at a time), and we stopped recruiting once we reached saturation. Developers were recruited through snowball sampling from various companies, while students were recruited from the university roster via

email. All participants had over three years of programming experience, regularly used LLM tools, and were compensated $40 via an Amazon gift card.

Demographics. We had three female and seven male participants. All of them have more than three years of experience in software development. Four participants worked on a Python project, and the other six worked on a web application project, which included the use case of HTML, CSS, and JavaScript languages. Six participants only used ChatGPT during the study session, two participants only used GitHub Copilot, and the other two used a combination of LLMs for the session. Details of participants are provided in Table II.

TABLE II
OBSERVATION STUDY PARTICIPANTS DEMOGRAPHICS

	Gnd.	Exp.	Langue(s).	LLM
OP1	F	3-6 yrs	HTML, CSS, JS	ChatGPT
OP2	M	3-6 yrs	HTML, CSS, JS	Copilot
OP3	M	>6 yrs	HTML, CSS, JS	Copilot, ChatGPT, Gemini
OP4	M	>6 yrs	Python	ChatGPT
OP5	F	3-6 yrs	Python	ChatGPT, Claude
OP6	M	3-6 yrs	HTML, CSS, JS	ChatGPT
OP7	M	3-6 yrs	HTML, CSS, JS	ChatGPT
OP8	F	>6 yrs	Python	ChatGPT
OP9	M	>6 yrs	HTML, CSS, JS	ChatGPT
OP10	M	>6 yrs	Python	ChatGPT

Coding Session. In the observational study, participants had 60-minute Zoom sessions where we recorded their screen and audio with consent while they interacted with AI code suggestions on self-chosen projects. We did not control the programming language to preserve natural behavior, and our analysis found no significant differences between languages. We tracked accepted or rejected suggestions and noted any modifications, clarifying ambiguous actions with follow-up questions. The sessions were transcribed, and all interactions, verbalizations, and prompts were logged. Two authors conducted descriptive qualitative coding [44] to categorize the reasons for accepting or rejecting suggestions, using trust factors from the survey, and analyzed code modifications or deletions instances through a negotiated agreement [45].

Post-study Interview. We conducted a 15-minute follow-up interview session where we asked participants about their prior good and bad experiences with code generation models and how that affected their trust. We also asked questions like "*Can you describe a specific experience when your trust in an AI code suggestion significantly increased?*" and, "*When you see a code suggestion, can you think of steps you go through to decide whether to accept/reject it?*" The complete set of questions is provided in supplementary materials[1]. To analyze, two researchers collaboratively performed a round of descriptive coding through negotiation along two dimensions: 1) annotating their reasons for trusting suggestions with the trust factors from the survey (closed coding) (e.g., "*[I] realize how it's working. It just has ten words, and it's guessing the 11th word.*" [P4] was annotated with the Comprehensibility factor), and 2) open coding their experiences with AI code assistants (e.g., "*If it gets it right, then it can handle my second task*" [P2] was coded with Positive Prior Expectation). Detailed coded answers are in supplementary materials.

[1]https://figshare.com/s/7ace13271c37bc3eadbf

C. Expert and Member Validation of Guidelines

From our mixed method study, we extracted four guidelines that developers and researchers can apply to improve the trustworthiness of code suggestions. To validate these guidelines, we conducted a follow-up survey with 15 participants, seven experts with research experience in software engineering and AI, and eight randomly selected participants from the initial survey (among 29 participants). Table III summarizes their expertise in the SE industry, research, and artificial intelligence. In this survey, we collected feedback on the guidelines' impact, practicality, and overall agreement. We also asked them to provide suggestions for improvement and propose additional guidelines to complement the set.

TABLE III
PARTICIPANT EXPERIENCE IN SOFTWARE ENGINEERING (SE), SOFTWARE RESEARCH (SR), AND ARTIFICIAL INTELLIGENCE (AI)

		SE	SR	AI			SE	SR	AI
Experts	EXP1	20	20	5	Members	M1 (P21)	6	3	0
	EXP2	24	22	20		M2 (P2)	3	0	2
	EXP3	0	0	7		M3 (P5)	6	1	2
	EXP4	14	14	8		M4 (P17)	1	0	3
	EXP5	8	20	0		M5 (P11)	1	0	4
	EXP6	5	5	4		M6 (P3)	4	0	0
	EXP7	5	5	8		M7 (P8)	6	1	0
						M8 (P8)	0	0	0

To analyze the responses, we summarized participants' feedback for each guideline and then revised the guidelines based on their suggestions, incorporating improvements and adding new applications and design opportunities.

IV. How do software developers define a trustworthy code suggestion? (RQ1)

To answer this RQ, we look into the factors that participants found impactful in influencing their trust decisions, as well as their own definitions of trustworthy code.

A. Software factors that impact trustworthiness

Despite the vast body of literature concerning trust in software development, no prior work has explicitly defined the term "trustworthy code". Inspired by software quality metrics (see Section III-A), we identified seven factors that are related to the trustworthiness of code and asked participants to rate the impact of these factors on their trust process on a Likert scale. The results are illustrated in Figure 1; Factors are listed on the y-axis, and the length of each box is proportional to the number of participants rated at the impact level.

Participants found *Correctness* and *Comprehensibility* to be the most impactful factors in determining trustworthiness, with 29 and 25 participants (out of 29) rating them as four or five (out of five), respectively. *Maintainability* and *Similarity* to the code base were rated substantially impactful with rated 11 and 16 times as substantial respectably. While *Security* was

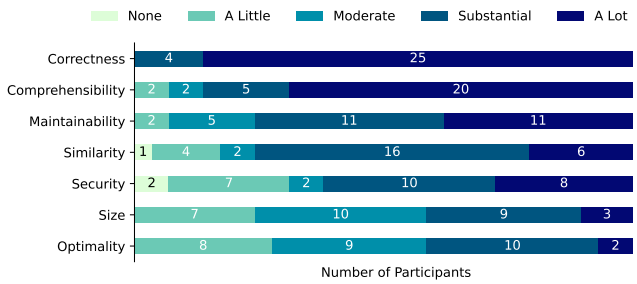

Fig. 1. From the survey: impact level of software quality factors on the trustworthiness of code.

also rated mostly substantial, the opinion about its impact on trustworthiness varied in the range.

While secure and optimal code might conventionally be considered the most critical criteria for trust in a code snippet, the participants from our study assigned little importance to either of them. Instead, they rated correctness and comprehensibility as the most significant factors influencing trust, with maintainability and similarity to existing code following closely.

B. Defining Trustworthy code

Participants also provided their own definitions for trustworthy code suggestions in the survey. Our qualitative coding, presented in Table IV, revealed that participants have various dimensions that define "trustworthiness". We discuss each of these categories below:

Correctness. 18 participants used correctness as a factor to define trustworthiness, using two different approaches. One group (eight participants) expressed the suggested code should match their expectations of the given task and "*meet [their] requirements*" [P6]. The other group (10 participants) identified correct code as code that is "*precise*" [P10], doesn't "*introduce errors*" [P21], and is executable [P12]. Determining functional correctness on the fly is non-trivial, especially when making a quick decision about whether to trust a code suggestion. Participants employed different strategies to assess correctness. Nine participants preferred testing code suggestions themselves, while two relied on tested code by other developers, e.g., P11: "*[I trust] when the code has been tested by [several] people over [some] scenarios*."

Comprehensibility. 16 participants mentioned comprehensibility as another factor to define trustworthiness. For example, P17 said they would trust a code suggestion depending on the ease of understanding: "*when [they] believe that [they] completely understand what the suggested code will do*". Similarly, P13 mentioned that "*a trustworthy code suggestion is [one that is] transparent or easy to understand*." Three participants [P8, P24, P26] mentioned they used code characteristics (*Comments* and *Variable Naming*) to determine comprehensibility. P8 stated they notice whether code has "*proper comments attached to its parts*" and "*understandable names*" before trusting the suggestion. Other participants [P2, P3] argued for *Simple Logic* as important for comprehension, as "*the code logic [should be] elementary*" to make it easier to comprehend [P2].

Maintainability. Five participants pointed out that the suggested code should be maintainable to trust and adapt to their code base. For example, P22 stated that a "*trustworthy code suggestion to me is when [. . .] the code is maintainable*." Participants also considered whether the code is easily adaptable when trusting a suggestion. For example, P14 considers "*a suggestion as a template that [they] can build upon and fix.*"

Five participants [P22, P26, P27, P28, P29] emphasized that adherence to *conventions*, including code and project standards, is crucial for trusting code. Participants [P12, P24, P29] preferred suggestions *similar to their code base* for improved cohesion and flow. Twelve participants mentioned additional factors: *robustness* [P12, P15], which should *handle all situations*", *minimal* code (P10) that is *to-the-point*" and *does not cause side effects*" [P3]. Other factors included trust in external dependencies and the code's educational value. Trust factors beyond the code were categorized as Personal and Foreign Factors. P9, identifying as *old school*", preferred traditional methods like asking *questions on Stack Overflow*" over trusting AI. *Positive Prior Experience* with the code suggestion model also influenced trust [P7], aligning with prior research [30], [46], [47]. As a foreign factor, task complexity was noted by some participants [P12, P14, P16, P23, P27], such as P16, who *will use [the code suggestion] only for simple things that did not require much thinking.*"

Participants differ in their definitions of trustworthy code and utilize various code characteristics to define what makes a code suggestion trustworthy. Most commonly, participants find a code trustworthy when it is **functionally correct**, **comprehensible**, and **compatible** with their existing code base.

V. What factors do developers use to assess trustworthiness? (RQ2)

A. Code Trustworthiness Evaluation

We observed that different code characteristics co-occurred with varying scores of trustworthiness. In Figure 2, we list these code aspects based on the trustworthiness of code snippets with box plots showing the score distribution of suggestions. The horizontal axis is the trustworthiness score, and the vertical axis lists the code characteristics.

The most correlated code characteristics with high trust were *Common/Generic/Typical Code*, *Safe and Secure Practices*, *Educational Value*, *Testing*, and *Meeting Expectations* while *Incomplete Code*, *Incorrect Code*, *Code-Doc-string Mismatch*, *Risk Associated With A Decision*, and *Under Descriptive Doc-string* showed strong correlations with lower scores. The definitions for these codes are provided in detail in the supplementary materials. To understand how significantly differently the code characteristics influence trustworthy score,

TABLE IV
TRUST DEFINITION QUALITATIVE CODING RESULTS

Trust Dimension	Factors Used in Trustworthiness Definition: Description [Participants Who Mentioned]	
Correctness	**Testing:** is tested or easily testable [P6,7,11,12,15,20,26,27,28]	**Expectation Match:** matches the expectations of the developer [P1,3,5,13,20,23,24,27]
	Error-free: does not have errors or minimizes errors [P10,19,21,22,25,26,28]	**Executable:** runs successfully [P12,13,19,21]
	Peer Reviewed: is reviewed by other developers [P11,26]	**Precise/Accurate:** is precise and accurate [P10]
	Requirement Match: matches the task's requirements [P6]	-
Comprehensibility	**Ease of Understanding:** is easily understood by the developer [P1,7,13,15,17,18,20,25,26,28]	**Comments:** has adequate/good comments [P8,24,26]
	Variable Naming: uses well-named variables [P8,10]	**Simple Logic:** has a simple logic [P2,3]
	Transparent: is transparent to the developer [P13]	**Code Familiarity:** is familiar to the developer [P4]
	Examples (I/O): provides input/output examples [P2]	-
Maintainability	**Templates:** serves as a good template/boilerplate [P14,16]	**Maintainability:** is easily maintainable [P22,26]
	Adaptability: is easily adaptable/changeable [P3,14]	-
Conventions	**Code Conventions:** adheres to general code/style conventions [P22,27,29]	**Project Conventions:** follows project-specific conventions [P26,28]
Similarity To Code Base	**Similarity To Code Base:** aligns with the existing codebase [P12,24,29]	**Cohesive:** is cohesive, similar to existing code [P29]
Robustness	**Robustness:** is robust to different conditions [P11,12]	**Corner Cases:** handles corner cases effectively [P5,25]
Minimality	**No Side-effect:** doesn't introduce side effects to the code base [P3,10,13,26]	-
Optimality	**Optimized:** is optimized [P25,26,28]	-
Improvement	**Improvement:** improves an aspect of the code [P26]	**Educational Value:** helps developers learn new things [P18]
Dependency	**3rd Party Library:** minimizes third-party library usage [P15]	-
Referenceability	**References:** has detectable training data references [P7]	-
Foreign Factors	**Task Complexity:** is influenced by task complexity [P12,14,16,23,27]	**Data Privacy Concerns:** addresses concerns about data leakage [P3]
	Code Complexity: is affected by code complexity [P29]	**Project Size Matters:** reflects the impact of project size [P3]
Personal Factors	**Old School:** developers prefer traditional approaches over AI [P9]	**Positive Prior Experience:** is influenced by past positive experiences [P7]

we perform an ANOVA across all characteristics, which revealed a significant difference ($F = 6.05$, $p = 5 \times 10^{-18}$) indicating at least one of the characteristics are different. To understand which pairs of characteristics are different, we perform Tukey HSD [48] with an adjusted p-value for multiple comparisons. We found that the impact of the five characteristics with the highest means (*Common/Generic/Typical code*, *Safe and Secure Practices*, *Educational Value*, *Testing*, and *Meeting Expectations*) on trustworthiness is significantly different from the impact of the five characteristics with the lowest means (*Under Descriptive Doc-string*, *Risk Associated With A Decision*, *Code-Docstring Mismatch*, *Incorrect Code*, and *Incomplete Code*) ($p_{adj} < 0.05$). Detailed comparison and significance tables are added in supplementary. *Generic code* was associated with the highest median trust score of nine out of ten, typically because "*it's a common function, and it's probable that it's correctly generated automatically*" [P5]. Notably, the lowest score this characteristic was observed with was seven, suggesting that generic code tends to contribute to participant trust positively. The second highest correlation was with *Secure Code*; when an element "*does not seem unsafe*" [P8] or the snippet showed "*good error handling*" [P2]. Additionally, participants were more likely to trust "*very informative code snippets*" [P6] as well as ones they would "*definitely test*" [P6]. Out of the top five most positively correlated characteristics, scores for codes that were *Meeting expectations* had by far the largest Inter-quartile range (IQR), with minimum, median, and maximum scores of 1, 7.5, and 10, respectively. While the code was sometimes rated higher because it "*clearly seems to accomplish the desired behavior*" [P13], participants sometimes also noted that they "do not fully understand" [P17] other parts of the snippet.

Code snippets identified as incomplete or corrupted had the lowest trustworthiness score. Participants became suspicious when there were missing parts of the code, and that negatively impacted their trust. For example, P3 said, "*the fact that the doc-string is incomplete make me very suspicious*." Similarly, in cases when they found the code is incorrect, like P2, who mentioned "*It looks wrong. It returns true if any of the name parts are longer than one character*", the trustworthiness score dropped sharply. The trustworthiness score was also low when the suggested code mismatched the doc-string. For example, P1 said, "*I don't trust the code [because it] actually doesn't do what the description says.*." Furthermore, when participants evaluated code snippets for riskier tasks, they were less trusting. As P1 said, "*I wouldn't trust any AI for anything public-key-cryptography related, out of personal experience*." In addition, descriptive doc-strings for the code snippets confused participants about the intended functionality, and that impacted their trust negatively. For example, P13 said, "*docstring are not very descriptive so I'm unsure what it does*."

Other code characteristics did not deterministically correlate with trustworthiness. Some of them have a wider IQR for trustworthiness. This means that the code snippets commented with those characteristics, like *Lack of Context* and *Incomprehensible Code*, were scattered further apart on the spectrum. Meanwhile, traits like *Bug-prone* and *No Need for AI* had averages and medians near the middle, suggesting little to no significant impact on trustworthiness.

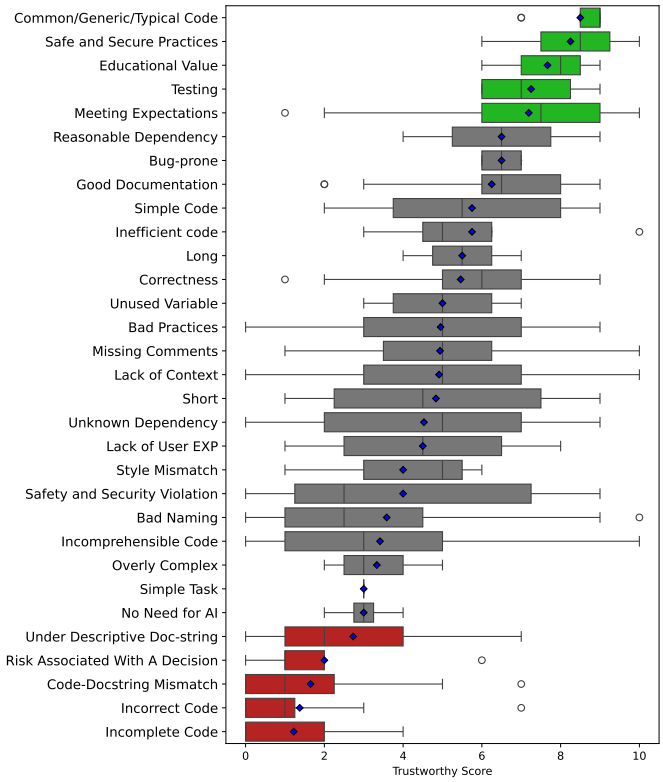

Fig. 2. Distribution of code snippets conditioned on code characteristics sorted by the descending order of average. Green bars are top five distribution with an average over 7, and red bars are bottom five those with an average lower than 3. Points are fliers, blue diamonds are means, and vertical lines are median. The box also shows the boundaries for first and third quartile.

There were code characteristics that correlated highly with high/low trustworthiness assessment. We found out that **Common/Generic/Typical Code** and **Safe and Secure Practices** deterministically correlate with trust most strongly while **Incomplete Code** and **Incorrect Code** deterministically correlate with distrust.

B. Definition Versus Assessment

Having identified the core characteristics correlated with code trustworthiness during evaluations, we sought to understand how the assessment process compared to the defined criteria. Did participants use the same factors to assess code trustworthiness as they did to define it?

We qualitatively coded participant comments on code snippets using the RQ1 codebook. Figure 3 highlights the frequency differences between factors used to define trustworthiness and those used to assess it. Frequencies were normalized by dividing each factor's count by the total factors in the respective coding sets. To understand the difference in the distribution of the trust factors between assessment and definition, we performed a χ^2 test. The result ($\chi^2 = 36.64$, $df = 6$, $p < 0.01$) revealed a significant difference between factors participants used during defining trustworthiness compared to when assessing it. Since the count for certain factors in assessment is zero, χ^2 leads to a p-value of 0. We replace factor counts 0 with $\epsilon \to 0$ to counter this. Although some trust factors were common in both definitions and assessments, certain factors were included in assessments but overlooked in definitions and vice versa.

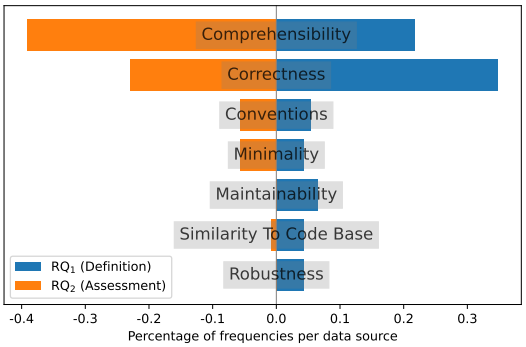

Fig. 3. Trust factors sorted by sum of relative frequencies in participants' definitions and assessments of trustworthiness. The blue bar represents the relative frequency of each factor in trust definitions, while the orange bar indicates its frequency in trustworthiness assessments.

Comprehensibility was the most frequent factor in participants' trust assessments, while *Correctness*, which was initially the most important in defining trust, was the second most frequent. We conclude that while participants thought they cared about *Correctness* when talking about trustworthiness, what they were looking for is the *Comprehensibility*. Furthermore, while *Dependency*, *Safety/Security*, and *Size* were frequently observed in participants' assessments of code snippet trustworthiness, they were ignored in participants' definitions of trustworthiness. Likewise, *Maintainability*, *Similarity To Code Base*, *Robustness*, and *Optimality* are mentioned in definitions but nearly never observed in the assessment.

Participants identified different factors when defining and assessing trustworthiness. While **Correctness** was most cited in definitions, **Comprehensibility** dominated assessments. Factors like **Dependency** and **Safety/Security** were considered in assessments but overlooked in definitions, while **Maintainability** was often mentioned in definitions but rarely assessed.

VI. Why do developers alter their trust in code suggestions? (RQ3)

Given the gap in the developers' ability to evaluate characteristics, we ask the following question:

A. How often do developers change trust decisions?

Across 10 observation sessions, participants generated 142 AI code suggestions. Participants initially accepted 82% (117) and rejected the rest. We consider a suggestion to be "rejected" if ignored. Of the 117 accepted, only 52% (75) remained in the code base by the end of the sessions (60-90 minutes each). Thus 48% (67) were ultimately not used—either immediately rejected, changed, or later removed (Figure 4).

We found that participants frequently accepted a code suggestion based on their perceived functional correctness (61/117). Nine out of ten participants, all except OP6, used testing to verify the correctness of the suggestions; for instance, OP5 asked ChatGPT for a data parser and tested it immediately before they accepted it. Others [OP2, OP8, OP9] evaluated whether the code met their expectations. For example, OP9, after requesting a detailed boilerplate from ChatGPT, found the suggestion satisfactory, noting, "*It looks good; it covers most of the parameters.*"

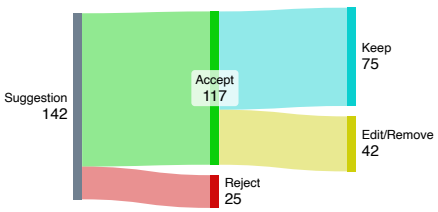

Fig. 4. Flow of developer trust decisions during the coding sessions.

B. What causes altered trust?

The gap between the high acceptance and low retention rates led us to investigate why developers trust particular suggestions. Two authors labeled each suggestion instance with trust factors extracted from RQ1 and RQ2. Here, we define altered trust as instances where a participant makes a trust decision they later change. To identify these, we characterize pairs (x, y) such that x represents the factor for which a suggestion was initially accepted, and y represents the factor for which it was eventually edited/deleted. Our analysis revealed three common reasons for altered trust.

(a) (Correctness, Correctness). In six instances, participants wrongly trusted (hereafter referred to as mistrusted) the correctness of a suggestion. Participants initially trusted a suggestion for its correctness but later discovered its flaws upon further analysis. For instance, OP3 asked ChatGPT to add a new tab to their application. The code seemed correct after testing but later led to an error.

(b) (Correctness, Minimality). In seven instances, participants overlooked an aspect of *Minimality* when first trusting a suggestion: They initially trusted it because it was correct but later realized that it needed to be edited or removed because it made unwanted changes. For example, OP8 accepted new CSS for his application, but he realized it changed the screen size, which he hadn't asked for.

(c) (Blind, Correctness). We observed 13 instances where participants "blindly" trusted suggestions that they eventually had to modify or remove from the code base entirely. By blindly accepting, we mean participants did not evaluate code suggestions before accepting. Of those, eight required modification because they were incorrect. We found that in only four cases, "blindly trusted" code stays in the code base.

Despite the challenges, our participants remained receptive to new code suggestions, even in the face of mistrust.

C. Attitudes about trust toward AI code suggestions

By asking some post-study interview questions, we wanted to explore more about how prior positive or negative experiences with AI affected developers' overall trust.

For the negative experiences, we examined the trust factors that led to acceptance and the reasons that they decided to modify the code further. We asked participants about situations where they felt confident in AI output, but it later turned out to be wrong, and how that affected their trust. Most participants (8 out of 10) reported that these negative experiences did not significantly impact their overall trust in the AI. They explained that they viewed AI suggestions as "*guidance, not a copy [source]*" [OP1] or said, "*I never blame the LLM; I blame the prompt*" [OP9]. This suggests that participants often forgive errors, seeing them as part of the tool's limitations rather than reasons to distrust it.

In contrast, positive experiences with AI increased participants' trust. Nine out of ten participants reported that their trust grew after good experiences. They were impressed by AI's ability to write complete code from scratch [OP8, OP9], debug [OP5], or resolve merge conflicts [OP6]. This enthusiasm shows how positive experiences bolstered their trust in AI. When participants were dissatisfied with the code, they often cited Comprehensibility [OP2, OP4, OP6] as a key factor. OP4 noted that the output can be misleading because these models "*just make everything sound so confident and so rosy. And, like, beautiful.*" Dissatisfaction frequently stemmed from issues with Correctness [OP1, OP2, OP3, OP4, OP5].

> Participants showed a positive bias towards the LLM. Their trust increases with good experiences but remains unaffected by negative ones. They take decisions by the **Comprehensibility** factor and edit because of **Correctness**.

Exploring the steps developers took to form their trust in AI code assistants, we asked, "*When you see a code suggestion, can you think of steps you go through to decide whether to accept/reject it?*" Some participants, like OP6, focused only on the size of the suggestion, avoiding anything "*more than 100 lines*". Others, such as OP7 and OP9, said they prioritized *Correctness*, with OP9 engaging in a back-and-forth dialogue with the AI to resolve errors. Some participants had more than one step to trust, evaluating comprehensibility first and then checked for dependencies.

> When considering the steps they go through to decide whether to trust a code, some participants decide solely based on **Comprehensibility**, **Correctness**, or code **Size**. Others start with one of them and examine other factors such as **Dependency** in the suggested code.

Finally, we note that the findings from the observation study validate those from the survey presented in RQ1 and RQ2. Consistent with the survey, we observed participants use comprehensibility and correctness as the most common factors when deciding which code to trust.

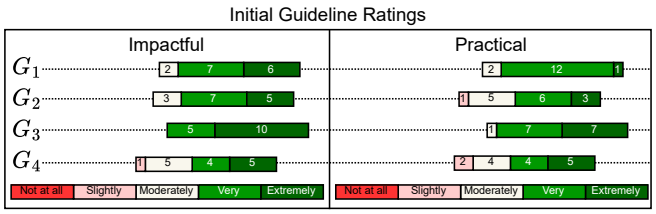

Fig. 5. Impactfulness and Practicality Ratings of Initial Guidelines: G_1: Clarification, G_2: Prioritize Code Quality, G_3: Evaluate Thoroughly, G_4: Value Simplicity. Most participants rated the guidelines as very or extremely impactful and practical.

VII. Guidelines for Improving Trustworthiness

We gathered four initial guidelines to help developers assess code trustworthiness and validated them with a survey of seven experts and eight original survey participants. The guidelines were rated as "*very reasonable*" [M8] by most, with many finding them impactful and practical (Figure 5). Participants suggested ways to enhance their application and design, leading to improved guidelines, which are detailed in this section (original guidelines are in the supplementary materials).

G_1**−Double-Sided Clarification:** Using structured prompts and precise language to define desired functionalities, reducing ambiguity and minimizing revisions—by using templates like "Consider these requirements: [R1], [R2]…". We found participants with clear requirements changed their code 22% less than those without. Developers should understand their software specifications, which is a standard expectation even without AI assistance. AI assistants can, in turn, ask clarifying questions to address ambiguities based on software specifications [49]. This helps by providing the problem context and generating more relevant code suggestions.

Application. Developers can explicitly define key requirements (function structure, expected outputs, data flow based etc.) in their prompts. Structured prompt frameworks such as EARS [50] or tools like DOORS [51] for feeding the code context can help organize these detailed specifications [EXP6]. Additionally, chain-of-thought reasoning [52] and logic verifiers can assist in evaluating whether the generated code matches high-level specifications [M4].

Design Opportunities. Developing prompt engineering techniques like adaptive templates [53] tailored to the specific project requirements can improve AI interactions. Tools can be designed to allow real-time refinement of prompts, e.g., LLMs asking clarifying questions during code generation. Additionally, integrating methods such as program analysis [54] [M6, M7], adding contextual project data through RAG, and developing a requirement coverage assessment mechanism [EXP6] can help improve interactions.

G_2**−Prioritize Code Quality Preferences:** Defining preferences for key code qualities—like comprehensibility, performance, and style—before engaging with AI code assistants is important. While functional requirements like correctness are commonly accounted for, specifying non-functional priorities helps evaluate suggestions and reduces revisions. For example, comprehensibility may be essential during the early exploratory coding phase [14], whereas maintainability may take precedence in the later post-deployment phases.

Application. Explicitly stating preferences in prompts—for example, requesting "readable code with comments and descriptive variable names" when comprehensibility is key helps ensure the generated code meets the desired criteria. They could request generated code to be compliant with specific coding guidelines, like MISRA-C [EXP2]. This guideline can be applied not only when prompting but also when evaluating that AI-generated code throughout the development cycle.

Design Opportunities. Incorporating software metrics, like Halstead metrics [25], alongside traditional NLP metrics [EXP5] into training or fine-tuning code-AI assistants can provide additional feedback for code generation. Creating tools for detecting code smells [EXP6], performing static analysis [55] [EXP2, M7], and comparing implementations using fuzzing technics [56] [EXP4] can further manage code quality. SMT solvers and memory checkers can improve the security of AI-generated code [EXP2].

G_3**−Evaluate Thoroughly:** Blindly accepting AI-generated code suggestions can lead to significant issues that require time-intensive fixes. This may occur when developers suspend critical thinking [18] due to anthropomorphizing such tools [57] or automation bias [58]. By treating AI-suggested code with skepticism [EXP2]—as contributions from an external source rather than self-authored—developers can identify and address potential flaws early, reducing redundant work and preventing the propagation of errors.

Application. Developers should treat AI-generated code skeptically (as with code from a third-party source) and apply thorough evaluation before acceptance. They should run thorough unit tests to verify correctness, especially for complex or high-risk implementations. Requesting chain-of-thought explanations can also clarify the logic behind AI suggestions, improving understanding, performance, and trust.

Design Opportunities. Future tools that help detect high-stakes scenarios and automatically generate unit tests to prevent edge case bugs can assist with evaluation. These tools can create test cases based on prompts before producing code. Current LLMs behind code-assistants show errant behavior when prompted with how epistemic markers for uncertainty [59]. Investigating how LLMs can understand and express uncertainty when the model is less confident [60] can help developers avoid over-trust and reliance, especially when the code is incorrect [M4]. Mechanisms for flagging security-critical or performance-intensive suggestions would help developers focus their evaluation efforts [M7].

G_4**−Value Simplicity:** Minimal and straightforward code is more reliable and easier to maintain. However, generative AI models often produce excessive code, adding unnecessary functions, variables, or modules that can conflict with requirements or introduce bugs [61]. Extra code increases the risk of redundancy, technical debt, and complications in future development with undefined elements, like hallucinated functions or variables. While prioritizing simple code is essential, avoiding

overly simplistic metrics, like lines of code, is important as they are poor proxies for true simplicity [M6]. We also found that participants preferred simpler AI-generated code

Application. Developers should evaluate AI-generated code for unnecessary features to prevent future issues like feature creep. If extra features are necessary, specific tests should be created to validate them [M8]. Any undefined entities, such as functions, variables, or libraries introduced by the AI, should be verified for accuracy and hallucination [EXP3].

Design Opportunities. Future tools should aim to reduce verbosity in code generation by identifying and removing unnecessary features [M8]. Reflection techniques like self-correction systems and self-consistency checks could evaluate outputs against the original prompt, ensuring no extra features or dependencies are added [M1]. Tools to map prompt requirements to specific code sections can further improve the traceability of added functionalities [EXP2].

VIII. Discussion

A. Leniency towards AI

In our study, while 82% of code suggestions were accepted, only 52% remained in the codebase unchanged. This high acceptance rate shows that developers try to work with AI suggestions, even when incomplete or imperfect. This tolerance differs from human pair programming, where a 48% error rate would be unacceptable for any software engineer [62]. Developers' trust in AI code assistants increases with good suggestions, as found in post-study interviews. The forgiving attitude may be because of automation bias, which stems from a lack of understanding of the inner workings of language models. Such bias could lead to over-trusting the model's suggestions, prompting developers to accept bad code. Further research into the effect of bias on how developers trust AI models is needed to integrate AI sustainably into the software development workflow. Additionally, tools that track the longitudinal effectiveness of the factors used for assessing the trustworthiness of suggestions help developers improve their trust over time.

B. Statistical and Quantitative Models of Trustworthiness

Automatically filtering out trustworthy code suggestions requires quantifying trust. One approach is to generate a statistically valid 'trust" score associated with each code snippet. We can achieve this using statistical guarantees that provide a quantified measure of confidence in a machine learning model's performance [63]. For a given error tolerance $\delta \in (0, 1)$, these guarantees ensure that $\text{Prob}(r \geq R_\delta) \geq 1 - \delta$ where r is the property of interest and R_δ is a lower bound for this property.

One technique that can provide such a statistical guarantee to assess AI code assistants' trustworthiness is conformal prediction [64]. Let r represent the trustworthiness of a code suggestion. Conformal prediction uses a calibration dataset of trust scores from n code suggestions, denoted by $r_1, \cdots, r_n$ (assuming $r_1, ..., r_n$ are independent and identically distributed). As a preliminary result, in our study, we used user-provided trust scores from the survey as r_is, where $n = 57$. For $\delta = 0.1$, R_δ is computed as 0.25. This suggests that any generated responses would have a trustworthiness score above 0.25 with a 90% probability. As this value gets higher, the model is considered more trustworthy. However, obtaining expert ratings is challenging for applying such a model. Unlike other aspects of LLMs' output, such as fairness [65], measuring trustworthiness is not straightforward. Therefore, new models should incorporate quantitative trustworthiness measures using proxies like correctness or other relevant software metrics.

C. Implications, Future Work, and Design Opportunities

Our findings and guidelines provide practical implications for developers, future research directions, and opportunities for tool builders to improve developer-AI trust dynamics.

Implications using generative AI. Our guidelines outline strategies for developers to improve interactions with AI code assistants. Emphasizing prompt clarity, evaluating code proactively, and favoring simplicity can improve the development process and reduce revisions. Developers must be aware of anthropomorphism bias (the tendency to ascribe human-like characteristics to things), reminding themselves that LLMs are merely tools and prone to hallucination. Managers can encourage developers to adopt project-specific templates to clarify requirements in prompts. Organizations should encourage placing specific tests/linters to check for common issues with AI-generated code (e.g., redundant code, dead code).

Future research for improving developer-AI interactions. In this study, we observed developers interacting with AI models during a one-hour session, allowing us to analyze short-term trust dynamics. In future research, we will extend these observations over longer periods to study how trust evolves and converges to specific approaches and usage patterns. Our study found that developers struggle to assess some code characteristics (e.g., maintainability, robustness) that are hard to assess on the fly. While software engineering research on maintainability or robustness has generated many verified metrics/estimations, they require analyzing the full codebase or running heavy computations. As developers adapt smaller snippets of AI-generated code, researchers can help identify how such characteristics can be estimated for small-sized snippets. Finally, further research is needed to investigate how user demographics like age, experience level, education, project types, and cultural background impact developers' trust. Automatically identifying personal trust factors and preferences can help tailor AI code assistants to provide suggestions that align with individual needs and trust levels.

Opportunities for Code-AI Tool Builders. Our results and guidelines indicate challenges and opportunities in building trustworthy human-AI interactions in software development. Our four guidelines $G_1 - G_4$ outline the specific needs of AI-code assistant models and researchers. Additionally, participants mentioned that adjusting the interaction level based on task complexity helps improve trustworthiness. For more complex tasks, iterative clarification questions could provide

more contextual suggestions, a similar approach explored for non-code tasks in [66].

D. Trustworthy Code Generation and Trusted AI

TABLE V
ALIGNMENT OF FINDINGS WITH EXISTING WORK

[Alignment] Comparison	Domain
✔ **Correctness** is critical for trust among developers. Matches how Autonomous systems' reliability and accuracy are core trust indicators.	TAI [67]–[69], HAI [70], MedAI [71]
✔ **Comprehensibility** was crucial among developers, similar to how trust is tied to explainability, transparency, and interpretability in AI systems.	TAI [72], [73], MedAI [74]
✔ **Generic code** boosts trustworthiness, aligns with familiarity, consistency, and similarity with prior outputs, enhances trust in AI.	TAI [75]–[77]
✖ **Safety and security** is overlooked in trust definitions but they are key to AI trustworthiness.	TAI [78]–[80]
✖ **Referenceability** is not prioritized by developers whereas citing sources is known to improve trust through traceability and accountability.	TAI [81]–[86]

TAI: Trusted AI, **HAI**: Human-AI Interaction, **MedAI**: Medical AI

While previous research in trustworthy AI and autonomous systems has concentrated mainly on enhancing AI's reliability and transparency, our work focuses on the user's perspective of assessing trustworthiness, specifically within software engineering. In this domain, trust dynamics remain underexplored. We found maintainability and similarity to codebase (RQ2) to be vital trust factors unique to software developers' trust dynamics. We also observed other unique characteristics developers use to decide the trustworthiness of code (RQ1), like educational value in code snippets (similar to [30]), reasonable dependency, unused variables, and code-docstring mismatch. This section compares our findings with related findings in AI and autonomous systems research (See Table V, ✔ shows alignment and ✖ shows contrast).

Alignment. Researchers in AI emphasize critical model qualities like reliability [67], [68] and accuracy [70], [71] as trust indicators similar to how developers in our study identified correctness as a trust factor [RQ1,2,3]. AI researchers show that the transparency of models improves trust as users can validate decisions [72], [74]. Our participants similarly prioritized the comprehensibility of the suggested code, using it as a proxy for transparent communication [RQ1,2,3]. AI researchers also found that familiarity with outputs [75], consistency [76], and similarity to previous experiences [77] enhance trust in AI interactions. This aligns with our observations that generic code snippets, likely familiar to participants, significantly boosted trustworthiness scores [RQ2].

Contradiction. One critical contrast is the prioritization of safety and security. Whereas AI ethics research highlights these as essential aspects of trustworthiness [78]–[80], developers in our study did not emphasize these aspects when defining trustworthy code [RQ2]. Additionally, AI tool development cites sources for their outputs to enhance trustworthiness [81], traceability [82], [86] and accountability [83]–[85]. However, we found referenceability (e.g., including citations) less important to developers in our study [RQ2].

IX. LIMITATIONS

Our study explores the cognitive process of trust in AI through rich qualitative insights. Like any qualitative study, it has some limitations. However, with 29 survey participants, 10 in the observational study, and 15 in expert checking, our participant count is comparable to or exceeds recent work [87], [88]. In our survey with 29 participants, we collected 486 trust ratings and comments on 200 code snippets. In the observation study with 10 participants, we analyzed 142 trust decisions. This variance allows us to conclude findings about how developers determine a code to be trustworthy. However, we acknowledge the limitations of our study in drawing definitive conclusions about inter-developer behavior patterns or the influence of developers' personalities on trust dynamics.

We made design decisions to scope our study to a feasible size. Therefore, our survey focused only on Python code, and the 60-minute observation session allowed us to observe only trust mechanisms within that time. To address these limitations, the observation study's findings (with participants using Python, JS, and HTML/CSS) triangulate with those from the survey, where participants had no time limit. Additionally, participants in the observation session used side-projects or backlogged code, which may not reflect the complexity of real-world product code. Future studies should explore trust mechanisms in organizational code as well.

X. CONCLUSION

This paper investigates how developers define, evaluate, and revise their trust in AI-generated code suggestions. Our mixed-methods study, including a survey (n=29) and an observation study (n=10), shows that trust is primarily based on correctness and comprehensibility, with maintainability also playing a role. However, developers lack tools to assess maintainability in real-time. We also found that developers often alter their trust decisions, keeping only 52% of the original suggestions. Based on these findings, we propose four guidelines to improve AI-developer interactions, which were validated by experts. These insights can inform the design of better AI code assistants and tools for human-AI pair programming.

ACKNOWLEDGMENT

This work is supported partially by a research gift from USC+Amazon Center on Secure and Trusted Machine Learning. We sincerely thank the reviewers for their valuable feedback and insights, which significantly improved this work. We are grateful to all the participants in our study, particularly Prof. Chao Wang, Prof. Mukund Raghothaman, Prof. William Halfond, Prof. Jieyu Zhao, Prof. Jyotirmoy Deshmukh, and Saad Shafiq for their guidance and support.

REFERENCES

[1] R. Anil, A. M. Dai, O. Firat, M. Johnson, D. Lepikhin, A. Passos, S. Shakeri, E. Taropa, P. Bailey, Z. Chen *et al.*, "Palm 2 technical report," *arXiv preprint arXiv:2305.10403*, 2023.

[2] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[3] S. Overflow, "AI | 2024 Stack Overflow Developer Survey." [Online]. Available: https://survey.stackoverflow.co/2024/ai/

[4] A. Khurana, H. Subramonyam, and P. K. Chilana, "Why and when llm-based assistants can go wrong: Investigating the effectiveness of prompt-based interactions for software help-seeking," in *Proceedings of the 29th International Conference on Intelligent User Interfaces*, 2024, pp. 288–303.

[5] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 754–768.

[6] A. Mastropaolo, L. Pascarella, E. Guglielmi, M. Ciniselli, S. Scalabrino, R. Oliveto, and G. Bavota, "On the robustness of code generation techniques: An empirical study on github copilot," *arXiv preprint arXiv:2302.00438*, 2023.

[7] D. N. Yaldiz, Y. F. Bakman, B. Buyukates, C. Tao, A. Ramakrishna, D. Dimitriadis, and S. Avestimehr, "Do not design, learn: A trainable scoring function for uncertainty estimation in generative llms," *arXiv preprint arXiv:2406.11278*, 2024.

[8] Y. Virk, P. Devanbu, and T. Ahmed, "Enhancing trust in llm-generated code summaries with calibrated confidence scores," *arXiv preprint arXiv:2404.19318*, 2024.

[9] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[10] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[11] V. Majdinasab, M. J. Bishop, S. Rasheed, A. Moradidakhel, A. Tahir, and F. Khomh, "Assessing the security of github copilot's generated code-a targeted replication study," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2024, pp. 435–444.

[12] A. Brown, S. D'Angelo, A. Murillo, C. Jaspan, and C. Green, "Identifying the factors that influence trust in ai code completion," in *Proceedings of the 1st ACM International Conference on AI-Powered Software*, 2024, pp. 1–9.

[13] R. Wang, R. Cheng, D. Ford, and T. Zimmermann, "Investigating and designing for trust in ai-powered code generation tools," in *Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency*, ser. FAccT '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1475–1493.

[14] S. Barke, M. B. James, and N. Polikarpova, "Grounded copilot: How programmers interact with code-generating models," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 85–111, 2023.

[15] C. Bird, D. Ford, T. Zimmermann, N. Forsgren, E. Kalliamvakou, T. Lowdermilk, and I. Gazit, "Taking flight with copilot: Early insights and opportunities of ai-powered pair-programming tools," *Queue*, vol. 20, no. 6, pp. 35–57, 2022.

[16] N. A. Ernst and G. Bavota, "Ai-driven development is here: Should you worry?" *IEEE Software*, vol. 39, no. 2, pp. 106–110, 2022.

[17] L. Gonçales, K. Farias, B. da Silva, and J. Fessler, "Measuring the cognitive load of software developers: A systematic mapping study," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 42–52.

[18] H. Mozannar, G. Bansal, A. Fourney, and E. Horvitz, "Reading between the lines: Modeling user behavior and costs in ai-assisted programming," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–16.

[19] F. F. Xu, B. Vasilescu, and G. Neubig, "In-ide code generation from natural language: Promise and challenges," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–47, 2022.

[20] A. Sarkar, A. D. Gordon, C. Negreanu, C. Poelitz, S. S. Ragavan, and B. Zorn, "What is it like to program with artificial intelligence?" *arXiv preprint arXiv:2208.06213*, 2022.

[21] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with ai assistants?" in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2785–2799.

[22] A. M. Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, and Z. M. J. Jiang, "Github copilot ai pair programmer: Asset or liability?" *Journal of Systems and Software*, vol. 203, p. 111734, 2023.

[23] J. Johnson, S. Lubo, N. Yedla, J. Aponte, and B. Sharif, "An empirical study assessing source code readability in comprehension," in *2019 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 2019, pp. 513–523.

[24] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative evaluation of software quality," in *Proceedings of the 2nd international conference on Software engineering*, 1976, pp. 592–605.

[25] C. Bailey and W. Dingee, "A software study using halstead metrics," in *Proceedings of the 1981 ACM workshop/symposium on Measurement and evaluation of software quality*, 1981, pp. 189–197.

[26] M. Wyrich, A. Preikschat, D. Graziotin, and S. Wagner, "The mind is a powerful place: How showing code comprehensibility metrics influences code understanding," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 512–523.

[27] C. E. C. Dantas and M. A. Maia, "Readability and understandability scores for snippet assessment: an exploratory study," *arXiv preprint arXiv:2108.09181*, 2021.

[28] B. Curtis, S. B. Sheppard, P. Milliman, M. Borst, and T. Love, "Measuring the psychological complexity of software maintenance tasks with the halstead and mccabe metrics," *IEEE Transactions on software engineering*, no. 2, pp. 96–104, 1979.

[29] D. G. Feitelson, "From code complexity metrics to program comprehension," *Communications of the ACM*, vol. 66, no. 5, pp. 52–61, 2023.

[30] B. Johnson, C. Bird, D. Ford, N. Forsgren, and T. Zimmermann, "Make your tools sparkle with trust: The picse framework for trust in software tools," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2023, pp. 409–419.

[31] H. Mozannar, G. Bansal, A. Fourney, and E. Horvitz, "When to show a suggestion? integrating human feedback in ai-assisted programming," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 9, 2024, pp. 10 137–10 144.

[32] "Code Quality Tool & Secure Analysis with SonarQube — sonarsource.com," https://www.sonarsource.com/products/sonarqube/, [Accessed 03-08-2024].

[33] M. Lacchia, "Various code metrics for python code," https://github.com/rubik/radon, [Accessed 03-08-2024].

[34] P. C. Q. Authority, "Bandit is a tool designed to find common security issues in python code." https://github.com/PyCQA/bandit, [Accessed 03-08-2024].

[35] "Effortless code quality and security for developers," https://www.codacy.com, [Accessed 03-08-2024].

[36] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[37] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner, "Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, 2023, pp. 654–668.

[38] G. Bhandari, A. Naseer, and L. Moonen, "CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software," in *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '21)*. ACM, 2021, p. 10.

[39] K. Srinath, "Python–the fastest growing programming language," *International Research Journal of Engineering and Technology*, vol. 4, no. 12, pp. 354–357, 2017.

[40] V. M. Ayer, S. Miguez, and B. H. Toby, "Why scientists should learn to program in python," *Powder Diffraction*, vol. 29, no. S2, pp. S48–S64, 2014.

[41] N. Nguyen and S. Nadi, "An empirical evaluation of github copilot's code suggestions," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 1–5.

[42] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.

[43] O. Vereschak, F. Alizadeh, G. Bailly, and B. Caramiaux, "Trust in ai-assisted decision making: Perspectives from those behind the system and those for whom the decision is made," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–14.

[44] K. J. Colorafi and B. Evans, "Qualitative descriptive methods in health science research," *HERD: Health Environments Research & Design Journal*, vol. 9, no. 4, pp. 16–25, 2016.

[45] R. Fisher, W. L. Ury, and B. Patton, *Getting to yes: Negotiating agreement without giving in*. Penguin, 2011.

[46] D. Ahn, A. Almaatouq, M. Gulabani, and K. Hosanagar, "Impact of model interpretability and outcome feedback on trust in ai," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–25.

[47] R. Cheng, R. Wang, T. Zimmermann, and D. Ford, ""it would work for me too": How online communities shape software developers' trust in ai-powered code generation tools," *ACM Transactions on Interactive Intelligent Systems*, vol. 14, no. 2, pp. 1–39, 2024.

[48] H. Abdi and L. J. Williams, "Tukey's honestly significant difference (hsd) test," *Encyclopedia of research design*, vol. 3, no. 1, pp. 1–5, 2010.

[49] B. Nuseibeh and S. Easterbrook, "Requirements engineering: a roadmap," in *Proceedings of the Conference on the Future of Software Engineering*, 2000, pp. 35–46.

[50] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak, "Easy approach to requirements syntax (ears)," in *2009 17th IEEE International Requirements Engineering Conference*. IEEE, 2009, pp. 317–322.

[51] T. Kuutti, "Comparing requirements management tools–ibm rational doors & hp alm," *Metropolia University of Applied Sciences*, 2019.

[52] J. Li, G. Li, Y. Li, and Z. Jin, "Structured chain-of-thought prompting for code generation," *ACM Transactions on Software Engineering and Methodology*, 2023.

[53] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, "A prompt pattern catalog to enhance prompt engineering with chatgpt," *arXiv preprint arXiv:2302.11382*, 2023.

[54] S. S. Muchnick and N. D. Jones, *Program flow analysis: Theory and applications*. Prentice-Hall Englewood Cliffs, 1981, vol. 196.

[55] B. Chess and G. McGraw, "Static analysis for security," *IEEE security & privacy*, vol. 2, no. 6, pp. 76–79, 2004.

[56] A. Takanen, J. D. Demott, C. Miller, and A. Kettunen, *Fuzzing for software security testing and quality assurance*. Artech House, 2018.

[57] M. Cohn, M. Pushkarna, G. O. Olanubi, J. M. Moran, D. Padgett, Z. Mengesha, and C. Heldreth, "Believing anthropomorphism: Examining the role of anthropomorphic cues on trust in large language models," in *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–15.

[58] K. Goddard, A. Roudsari, and J. C. Wyatt, "Automation bias: a systematic review of frequency, effect mediators, and mitigators," *Journal of the American Medical Informatics Association*, vol. 19, no. 1, pp. 121–127, 2012.

[59] K. Zhou, D. Jurafsky, and T. Hashimoto, "Navigating the grey area: How expressions of uncertainty and overconfidence affect language models," *arXiv preprint arXiv:2302.13439*, 2023.

[60] M. Xiong, Z. Hu, X. Lu, Y. Li, J. Fu, J. He, and B. Hooi, "Can llms express their uncertainty? an empirical evaluation of confidence elicitation in llms," *arXiv preprint arXiv:2306.13063*, 2023.

[61] K. Saito, A. Wachi, K. Wataoka, and Y. Akimoto, "Verbosity bias in preference labeling by large language models," *arXiv preprint arXiv:2310.10076*, 2023.

[62] A. Begel and N. Nagappan, "Pair programming: what's in it for me?" in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, 2008, pp. 120–128.

[63] J. Lederer, "Statistical guarantees for sparse deep learning," *AStA Advances in Statistical Analysis*, pp. 1–28, 2023.

[64] G. Shafer and V. Vovk, "A tutorial on conformal prediction." *Journal of Machine Learning Research*, vol. 9, no. 3, 2008.

[65] A. Fayyazi, M. Kamal, and M. Pedram, "Facter: Fairness-aware conformal thresholding and prompt engineering for enabling fair llm-based recommender systems," *arXiv preprint arXiv:2502.02966*, 2025.

[66] F. Mu, L. Shi, S. Wang, Z. Yu, B. Zhang, C. Wang, S. Liu, and Q. Wang, "Clarifygpt: Empowering llm-based code generation with intention clarification," *arXiv preprint arXiv:2310.10996*, 2023.

[67] E. Toreini, M. Aitken, K. Coopamootoo, K. Elliott, C. G. Zelaya, and A. Van Moorsel, "The relationship between trust in ai and trustworthy machine learning technologies," in *Proceedings of the 2020 conference on fairness, accountability, and transparency*, 2020, pp. 272–283.

[68] M. Ryan, "In ai we trust: ethics, artificial intelligence, and reliability," *Science and Engineering Ethics*, vol. 26, no. 5, pp. 2749–2767, 2020.

[69] J. R. Schoenherr, *Ethical artificial intelligence from popular to cognitive science: Trust in the age of entanglement*. Routledge, 2022.

[70] J. R. Schoenherr, R. Abbas, K. Michael, P. Rivas, and T. D. Anderson, "Designing ai using a human-centered approach: Explainability and accuracy toward trustworthiness," *IEEE Transactions on Technology and Society*, vol. 4, no. 1, pp. 9–23, 2023.

[71] A. J. London, "Artificial intelligence and black-box medical decisions: accuracy versus explainability," *Hastings Center Report*, vol. 49, no. 1, pp. 15–21, 2019.

[72] A. Ferrario and M. Loi, "How explainability contributes to trust in ai," in *Proceedings of the 2022 ACM Conference on Fairness, Accountability, and Transparency*, 2022, pp. 1457–1466.

[73] X. Li, H. Xiong, X. Li, X. Wu, X. Zhang, J. Liu, J. Bian, and D. Dou, "Interpretable deep learning: Interpretation, interpretability, trustworthiness, and beyond," *Knowledge and Information Systems*, vol. 64, no. 12, pp. 3197–3234, 2022.

[74] J. Fehr, G. Jaramillo-Gutierrez, L. Oala, M. I. Gröschel, M. Bierwirth, P. Balachandran, A. Werneck-Leite, and C. Lippert, "Piloting a survey-based assessment of transparency and trustworthiness with three medical ai tools," in *Healthcare*, vol. 10, no. 10. MDPI, 2022, p. 1923.

[75] G. Kučinskas, "Negative effects of revealing ai involvement in products: Mediation by authenticity and risk, moderation by trust in ai and familiarity with ai," *Moderation by Trust in Ai and Familiarity with Ai*, 2024.

[76] P. K. Kahr, G. Rooks, M. C. Willemsen, and C. C. Snijders, "Understanding trust and reliance development in ai advice: Assessing model accuracy, model explanations, and experiences from previous interactions." *ACM Transactions on Interactive Intelligent Systems*, 2024.

[77] M. Yi and H. Choi, "What drives the acceptance of ai technology?: the role of expectations and experiences," *arXiv preprint arXiv:2306.13670*, 2023.

[78] N. Gillespie, S. Lockey, C. Curtis, J. Pool, and A. Akbari, "Trust in artificial intelligence: A global study," *The University of Queensland and KPMG Australia*, vol. 10, 2023.

[79] B. Stanton, T. Jensen *et al.*, "Trust and artificial intelligence," *preprint*, 2021.

[80] M. Mylrea and N. Robinson, "Artificial intelligence (ai) trust framework and maturity model: applying an entropy lens to improve security, privacy, and ethical ai," *Entropy*, vol. 25, no. 10, p. 1429, 2023.

[81] B. Mittelstadt, C. Russell, and S. Wachter, "Explaining explanations in ai," in *Proceedings of the conference on fairness, accountability, and transparency*, 2019, pp. 279–288.

[82] A. Holzinger, "The next frontier: Ai we can really trust," in *Joint European conference on machine learning and knowledge discovery in databases*. Springer, 2021, pp. 427–440.

[83] H. Liu, Y. Wang, W. Fan, X. Liu, Y. Li, S. Jain, Y. Liu, A. Jain, and J. Tang, "Trustworthy ai: A computational perspective," *ACM Transactions on Intelligent Systems and Technology*, vol. 14, no. 1, pp. 1–59, 2022.

[84] A. Duenser and D. M. Douglas, "Who to trust, how and why: Untangling ai ethics principles, trustworthiness and trust," *IEEE Intelligent Systems*, 2023.

[85] D. Kaur, S. Uslu, K. J. Rittichier, and A. Durresi, "Trustworthy artificial intelligence: a review," *ACM computing surveys (CSUR)*, vol. 55, no. 2, pp. 1–38, 2022.

[86] R. Huang and S. Chattopadhyay, "A tale of two communities: Exploring academic references on stack overflow," in *Companion Proceedings of the ACM on Web Conference 2024*, 2024, pp. 855–858.

[87] E. J. Arteaga Garcia, J. F. Nicolaci Pimentel, Z. Feng, M. Gerosa, I. Steinmacher, and A. Sarma, "How to support ml end-user programmers through a conversational agent," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.

[88] M. Sánchez-Gordón, R. Colomo-Palacios, and A. Sanchez Gordon, "Characterizing role models in software practitioners' career: An interview study," in *Proceedings of the 2024 IEEE/ACM 17th International Conference on Cooperative and Human Aspects of Software Engineering*, 2024, pp. 164–169.