HILDE: Intentional Code Generation via Human-in-the-Loop Decoding

Emmanuel Anaya González*
UC San Diego
fanayagonzalez@ucsd.edu

Raven Rothkopf*
UC San Diego
rrothkopf@ucsd.edu

Sorin Lerner
UC San Diego
lerner@ucsd.edu

Nadia Polikarpova
UC San Diego
npolikarpova@ucsd.edu

Fig. 1: HILDE: an assistant that A *highlights critical decision points* in an LLM code completion, B displays *local alternatives* the model considered at a particular decision point, C *explains differences* between these alternatives, and D lets users select a preferred alternative over the original completion, aligning code generation with their personal goals.

***Abstract*—While AI programming tools hold the promise of increasing programmers' capabilities and productivity to a remarkable degree, they often exclude users from essential decision-making processes, causing many to effectively "turn off their brains" and over-rely on solutions provided by these systems. These behaviors can have severe consequences in critical domains, like software security. We propose *Human-in-the-Loop Decoding*, a novel interaction technique that allows users to observe and directly influence LLM decisions during code generation, in order to align the model's output with their personal requirements. We implement this technique in HILDE, a code completion assistant that *highlights critical decisions* made by the LLM and provides *local alternatives* for the user to explore. In a within-subjects study (N=18) on security-related tasks, we found that HILDE led participants to generate significantly fewer vulnerabilities and better align code generation with their goals compared to a traditional code completion assistant.**

***Index Terms*—Human-AI Collaboration, Program Synthesis, AI Programming Assistants, Software Security**

I. INTRODUCTION

As AI-powered programming tools are integrated into everyday development workflows, programmers increasingly sacrifice their autonomy for perceived productivity. Traditionally, programmers engaged in continuous, intentional decision-making during the coding process, choosing implementation strategies based on their specific non-functional requirements (i.e, security, efficiency, readability), company policies, and personal preferences. LLMs have now automated this process, obscuring many subtle choices and presenting only a single "best" solution based on inscrutable statistical patterns.

AI's automation of high and low-level implementation decisions has significant consequences: First, programmers remain **unaware** of alternative strategies that may be more objectively correct, contextually appropriate, or personally aligned [1]. Second, programmers **over-rely** on AI outputs, blindly trusting them to be correct without sufficient comparison with alternatives [2], [3]. Third, programmers lack the **agency** to effectively control LLM behavior, relying on vague prompt-tuning to steer code generation instead of selecting or composing a unique solution from a set of possible strategies [4].

The downstream consequences are most notable in critical domains like software security, where studies have found that programmers write less secure code with AI assistants than without [5]–[7]. LLMs frequently generate code that reflects popular but insecure practices from their training data [8], and are unable to account for newly discovered vulnerabilities [9], [10]. As a result, programmers may unwittingly accept insecure suggestions, mistaking model confidence for correctness.

To overcome these pitfalls, researchers have called for AI-resilient interfaces [11] that reengage users in decision-making by letting them choose from multiple AI suggestions [4], [12], [13]. However, with mainstream AI programming tools, users have to reverse-engineer AI choices from a set of alternative completions that lack visual cues of meaningful differences [3]. We offer a new approach: involve users directly in the model's fine-grained decision-making process, also called *decoding*. As an LLM generates a program, it predicts the next piece of code—or token—from a learned list of potential tokens, each with an associated probability [14], [15]. Each token in the list could transform code style, structure, or semantics, but end-users remain unaware of these considerations when they can only view the top tokens that are returned after decoding.

We introduce ***Human-in-the-Loop Decoding*, a novel interaction technique that enables programmers to directly influence LLM decision-making during code generation**.

*These authors contributed equally to this work.

Human-in-the-Loop Decoding exposes lower-probability options that a model might otherwise discard, enabling programmers to discover a richer variety of alternatives and select tokens that are more aligned with their intentions, rather than being limited to the model's most likely suggestions.

To realize Human-in-the-Loop Decoding, we developed HILDE[1]: a programming assistant (illustrated in Fig. 1, detailed in Sec. III) that adds two affordances on top of traditional code completion tools, such as GITHUB COPILOT [16]. First, HILDE *highlights critical decision points* in the LLM-generated code (A). Second, through a keyboard or mouse short-cut, HILDE can provide *local alternatives* for the user to explore (B), annotated with explanations of their differences (C). To determine the critical decision points, we combine the model's uncertainty [17]–[19] in a token with semantic information derived from analyzing alternative completions.

We ran a controlled user study (N=18) where participants completed security-related programming tasks using HILDE versus a baseline AI assistant (*c.f.* Sec. IV). We found that using HILDE, participants generated code with **significantly fewer vulnerabilities** despite a general lack of training in software security practices (*c.f.* Sec. V). Participants also used HILDE to catch and correct significantly more vulnerabilities than with the baseline. Additionally, HILDE helped users explore alternative strategies and reflect on what outcomes they actually wanted——often discovering their own intentions in the process. With this deeper understanding, they could steer code generation to directly align with their programming goals.

The contributions of this paper are as follows:

- *Human-in-the-Loop Decoding*, a novel interaction technique for intentional code generation with LLMs.
- HILDE: a code completion assistant equipped with two new affordances—*critical decision highlighting* and *local alternatives*—to promote Human-in-the-Loop Decoding.
- An *evaluation* of HILDE in a user study comparing participants' experiences using HILDE to a baseline assistant during security-related coding tasks.

II. RELATED WORK

A. AI Programming Assistants in Software Security

A growing body of research has examined the impact of AI-powered programming assistants on code security [20]. Most of this work focuses on improving the security of LLM generated code [21]–[23], and using LLMs to patch existing vulnerabilities [24]–[26] In contrast, we use security as a representative domain to investigate how interaction design influences programmers' awareness, agency, and intent when using LLMs——insights that may generalize to other real-world programming challenges.

Our study design draws the most direct inspiration from Perry et al.'s empirical evaluation of programmers using GITHUB COPILOT to complete isolated security-driven programming tasks [5]. Related studies confirm their findings that programmers working with LLMs frequently wrote less secure code than those without such assistance, often reproducing vulnerabilities or relying on insecure suggestions surfaced by the model [6], [7], [27].

[1]The name HILDE is short for "Human-in-the-Loop Decoding ".

B. Steering LLMs

Beyond correctness and security, preserving programmers' agency and intent is a core challenge for modern AI coding assistants [4], [28]–[30]. Research has shown that without sufficient guardrails, programmers tend to accept model suggestions without critical comparison [1]–[3].

Recent work has begun to address this by introducing systems for steering that help users refine and clarify intent throughout the programming workflow [13], [31], [32]. For instance, Kazemitabaar et al.'s work on promoting interactive task decomposition [33] via "phase-wise" and "step-wise" levels of interaction with AI programming assistants. Their "step-wise" approach offers a similar level of granularity as our local alternatives by providing intervention points at each step of solving a programming task and exposing editable LLM assumptions about the generated code. However, their method does not ensure a direct relationship between user edits and the resulting code, leaving users with limited control if the model fails to reflect their intent in the final output.

C. Visualizing AI Variance

Users are often unaware of the range of possible code suggestions that can be obtained from an LLM [3], [12], [13], [34]. Recent work sees this as a consequence of the current interaction paradigm prioritizing rapid development over exploration of the latent design space [34], [35]. Alternative visualizations [13], anchored explanations [36], [37], and structured interfaces supporting choices [4], [12] have been shown to help programmers reason about such sets of options.

Our work extends these efforts by displaying a rich array of alternatives and contextual explanations in a compact interface, to help programmers quickly make sense of the full spectrum of possible implementations.

D. Highlighting Uncertainty in LLM Code Generation

Several studies have proposed inline highlights as an interface to communicate uncertainty in LLM-generated code [19], [38], [39]. Vasconcelos et al. [17] found that decoding uncertainty highlighting alone offered no advantage over a baseline without it, underscoring the need for richer methods to expose uncertain code sections to the programmer. Kim et al. [40] show that natural language expressions of uncertainty are effective in reducing overreliance on LLM responses, and that assertive has a significant impact.

Our approach enhances token uncertainty highlighting by using semantic significance of the alternative as predicted by the model itself, as well as by including human-readable explanations about such alternatives at different levels of detail. This enables programmers to immediately explore and understand the effects of fine-grained implementation choices.

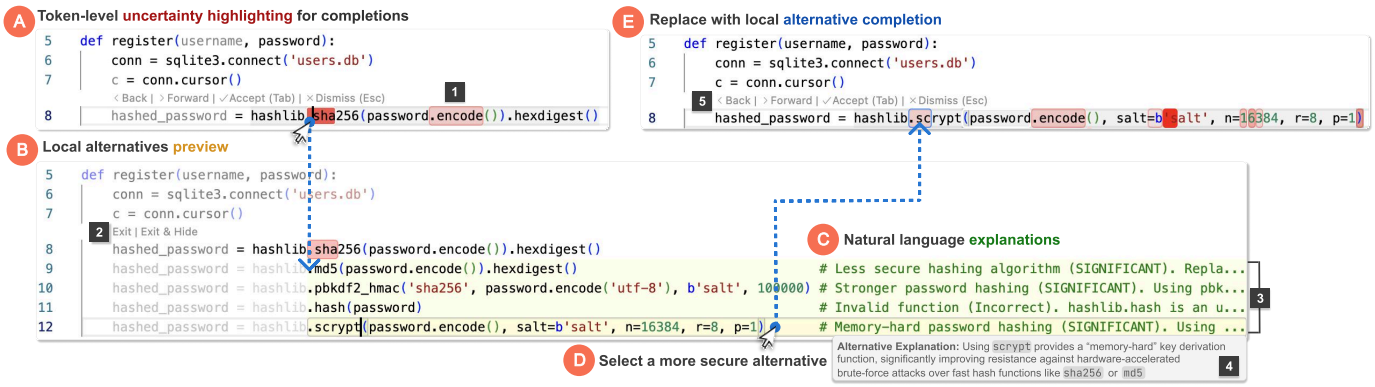


Fig. 2: HILDE is a VSCODE extension that visualizes an LLM's token-level uncertainty and encourages interactive exploration of code completions: (A) Users prompt the model via comments and/or code context. As code is generated, HILDE highlights tokens where the model is uncertain in a red gradient (1)—-darker indicates higher uncertainty. (B) Press `ENTER` to preview local alternative completions that the model was considering at that step, ordered by likelihood. (2) `Exit` (or moving the cursor away) closes the preview, whereas `Exit&Hide` disables further highlighting for that token. (C) HILDE provides a natural language explanation of how each alternative differs from the original as a truncated comment (3) or a more detailed tooltip on hover (4). (D) Replace an original token with an alternative by pressing `ENTER`. (E) HILDE regenerates subsequent code to reflect the new token (now with blue border). Navigate token edit history with `Back` and `Forward` codelenses (5), and accept or dismiss the completion at any time.

III. THE HILDE PROGRAMMING ASSISTANT

We implemented one concrete instantiation of our Human-in-the-Loop Decoding technique in a programming assistant called HILDE. This section demonstrates HILDE via a usage example and then describes its implementation.

A. HILDE *by Example*

Klaus, a graduate student, is building a web application to manage a mentoring program for junior researchers. Klaus uses Python regularly for his research, but he is new to web development, so he decides to use HILDE, an AI programming assistant, to help him write the code. Klaus needs to implement the function `register`, which registers a new user with a username and password; he starts by writing the function signature and then presses `CMD/CTRL+I` every time he wants to prompt HILDE to complete the next piece of its implementation.

(A) When HILDE suggests a completion on line 8, which hashes the password before storing it in the database, Klaus is about to accept the suggestion without a second thought, as it looks reasonable. However, he notices that HILDE highlighted some of the completion tokens in red, indicating that the model was uncertain about them.

(B) Klaus clicks on the first highlighted token, which happens to be the (beginning of the) hash function name. This brings up a list of alternative completions that the model considered at that step, which include four other hash functions available in the `hashlib` library.

(C) HILDE also shows comments with (truncated) explanations of how each alternative differs from the original. By skimming the comments, Klaus realizes that the different hash functions have different cryptographic strength; he hovers his mouse over two promising alternatives, `pbkdf2_hmac` and `scrypt`, and reads the full explanation that appears in the tooltip.

(D) Based on the needs of his application, Klaus decides to use `scrypt` and clicks on it to replace the original completion with this alternative.

(E) The new completion has a blue border around the choice point, allowing Klaus to come back and reconsider the choice later.

Since the new completion also has some highlighted tokens, Klaus explores alternatives for those as well (not shown in the figure). After bringing up the alternatives for the `encode` function, he decides that the default completion is good enough, and clicks on `Exit&Hide` (B) 2 to remove the highlighting for that token, so that he can focus on other critical choices. On the other hand, for the `salt` parameter, he decides to use random bytes instead of a constant string, once HILDE points out that the former is more secure.

B. *Implementation*

We implemented HILDE as a Visual Studio Code extension. The HILDE architecture is shown in Fig. 3, which references the same example code from Fig. 2 and Fig. 1.

1) Local alternatives explanations: To assess the impact of each local alternative, HILDE prompts the analysis LLM with the base completion, top token, and local alternative, asking it to return the following information as structured output[2]:

- *Detailed explanation:* An analysis of how the change would influence the current completion. (i.e. new parameters, control flow, libraries or side effects.)
- *Explanation summary:* A minimal description of the change, which must be understandable at a glance.

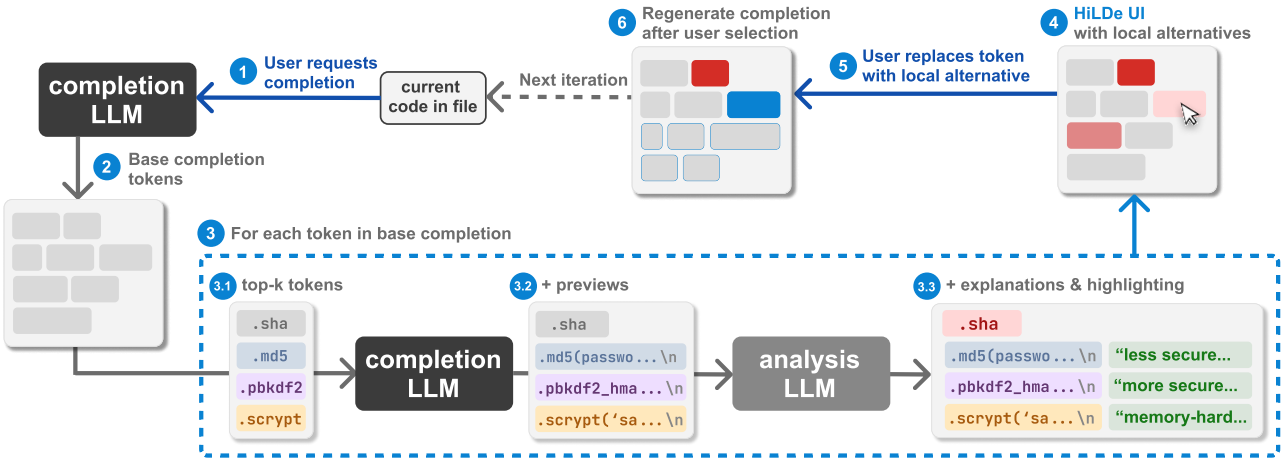


Fig. 3: The HILDE architecture. When a user requests a code completion 1, HILDE sends the user's prompt to the completion LLM, receiving a base completion and the top-k tokens (with probabilities) at each generation step 2. For each alternative token 3.1 at every step, HILDE asks the same completion LLM to generate a code preview (up to the next line break) showing what the completion would look like with that token 3.2. For each preview, HILDE queries the analysis LLM for an explanation of how the alternative differs from the original completion, and whether it would yeild *significant* changes to the code 3.3. HILDE then highlights the tokens with significant local alternatives in the editor 4. If the user selects a local alternative 5, HILDE completes it to a full snippet and updates the editor accordingly 6.

- *Category:* One of "Significant", "Minor" or "Incorrect". "Significant" changes considerably affect the behavior of the program, (i.e code security, efficiency, robustness, etc.) "Minor" changes are purely stylistic, like variable renaming, and do not affect the program's execution. "Incorrect" changes would result in invalid code, (i.e. syntax errors, invalid function calls, or non-existent libraries).
- *Importance Score* A float in the range $[0, 1]$ indicating the severity of the impact of this change. We use this when computing highlighting of critical steps.

We build explanation comments C 3 for each local alternative using the *Explanation summary* and *Category*. The user can view the *Detailed explanation* on hover C 4.

2) Uncertainty highlighting: The main goal of highlighting completion tokens is to draw attention to steps were the model made a *critical decision* that may need verification. One straightforward way to do so is by considering the model's internal uncertainty—since the LLM defines a probability distribution over tokens at each generation step, it is natural to consider the *entropy* of such distribution for highlighting. Related work finds this approach inadequate in practice [17].

First, the model regularly assigns high entropy to steps that are not critical for the user, (i.e changing a variable name or a debug message). As a result, a large fraction of steps are highlighted. In early pilots, we found that a completion with too much highlighting is overwhelming for users.

Second, the model often assigns low entropy to steps where the top token is actually incorrect, or should at least be verified by the user. For instance, when calling popular library functions that are prevalent in the training data, but are known to be inadequate specific use cases.

We use the explanations of each token's local alternatives to analyze the importance of each step more accurately 3.3. We define a new *corrected entropy* by updating the probability of each of the alternative tokens proportional to the *Importance Score* of the step. With this technique, steps with local alternatives that only lead to minor changes have low corrected entropy, whereas steps with at least one significant change are highlighted to the user. By tuning the weight of *Importance Score* on the corrected entropy, HILDE consistently highlights only a handful of critical decision points.

3) Completion regeneration with local alternatives: If a user replaces a token in the base completion with a local alternative token 5, the completion LLM generates a full code snippet from the new token to reflect the change 6. Given the stochastic nature of LLMs, the suffix of this alternative generation is *not* guaranteed to be equivalent to the base completion. We found this to be a cognitive hurdle for users in our pilots, since they did not expect the change to have downstream effects on their code. We address this by generating 10 different suffixes, keeping the one most similar [41] to the base completion. Though this does not guarantee that the suffix will be unchanged, it reflects the user's intuition.

4) Technical details: For the completion LLM, we use `Qwen2.5-Coder-32B` [42], an open-source state-of-the-art code model that supports fill-in-the-middle completions, and serve it via vLLM [43] using 2x Nvidia A100 40GB. For the analysis LLM, we use the smaller, faster `gpt4.1-nano` model to minimize latency and ensure a smooth user experience. For both models we set `temperature = 0` to ensure reproducibility of outputs across participants, and `max_tokens = 1024`. The rest of the parameters are set to their default values.

IV. METHOD

In this section, we describe how we designed our study[2], including the participant pool, procedure, data collection, and tasks to answer three core research questions:

1) Do users write **fewer security bugs** using HILDE?
2) How does HILDE affect **overreliance** on AI programming assistants?
3) How does HILDE affect users' ability to **reach their programming goals** when using LLMs?

A. Participants

Our goal was to recruit participants with a wide variety of programming experiences, but with a focus on those who used AI programming assistants regularly. Explicit knowledge of security principles was not a requirement for our study, though we did require our participants to have at least some familiarity with both Python and C. We recruited 18 participants, 9 self-identified as men, 8 as women, and 1 as non-binary. 3 were undergraduate students, 7 were masters students, 6 were Ph.D. students, and 2 were professional software engineers. All participants reported high levels of experience with Python: either advanced or expert. All participants reported at least moderate experience with C. All participants reported that they used AI programming assistants a few times a week or more.

B. Baseline

For the baseline, we reimplemented the inline completion feature from GITHUB COPILOT [16]. We opted for this, rather than using the official tool, to use the same underlying completions LLM as HILDE, rather than a closed-source model, and ensure a more reliable comparison. In BASELINE, the user can view, individually, up to five different completions for every prompt, and keep one if they choose to.

C. Tasks

Participants completed four programming tasks during our study—two tasks in Python and two in C—that presented a broad range of security challenges. We compiled tasks from related work on security and LLMs [5], [6], that were designed to be self-contained and solvable with a few lines of code. We use the same criteria as these prior research to assess the security level of solutions. For each task, participants were given a code skeleton that included a task description, and function signature, as well as a way to run and test the basic functionality of their code. This setup was intended to minimize the amount of time spent implementing boilerplate code and allow participants to focus on their interactions with the programming assistant and potential security considerations. The tasks were as follows:

1) SQL secrets (Python): Given a username and password, hash the password, generate private and public ECDSA keys, then store the username, password, and private key in a SQL database. Return the public key.

[2]For reproducibility, all study materials are available via OSF

2) Sandboxed directory (Python): Given a string `path` to a file, verify that the file is within the `/safedir` directory, then parse it as YAML and return its contents.

3) CSV file write (C): Generate 3 random floats, convert them to strings, then write them to a CSV file `data.csv`.

4) User input to struct (C): Query the user for a name, allocate a new `person` struct with this name and `status` set to 0, then print the struct's contents.

To ensure each task could be completed within 15 minutes, they were designed to be solvable within four interactions with the LLM. We specifically chose tasks and prompts that were likely to naturally cause the model to generate code with vulnerabilities, including unsafe memory management (T3, T4), arbitrary code execution (T1, T2), path traversals (T2), and insecure cryptographic libraries and algorithms (T1).

D. Procedure

We conducted the studies over Zoom, and participants completed tasks using the Visual Studio Code IDE in an isolated Github Codespaces environment [44]. Each participant received a $35 gift card upon completion of their study.

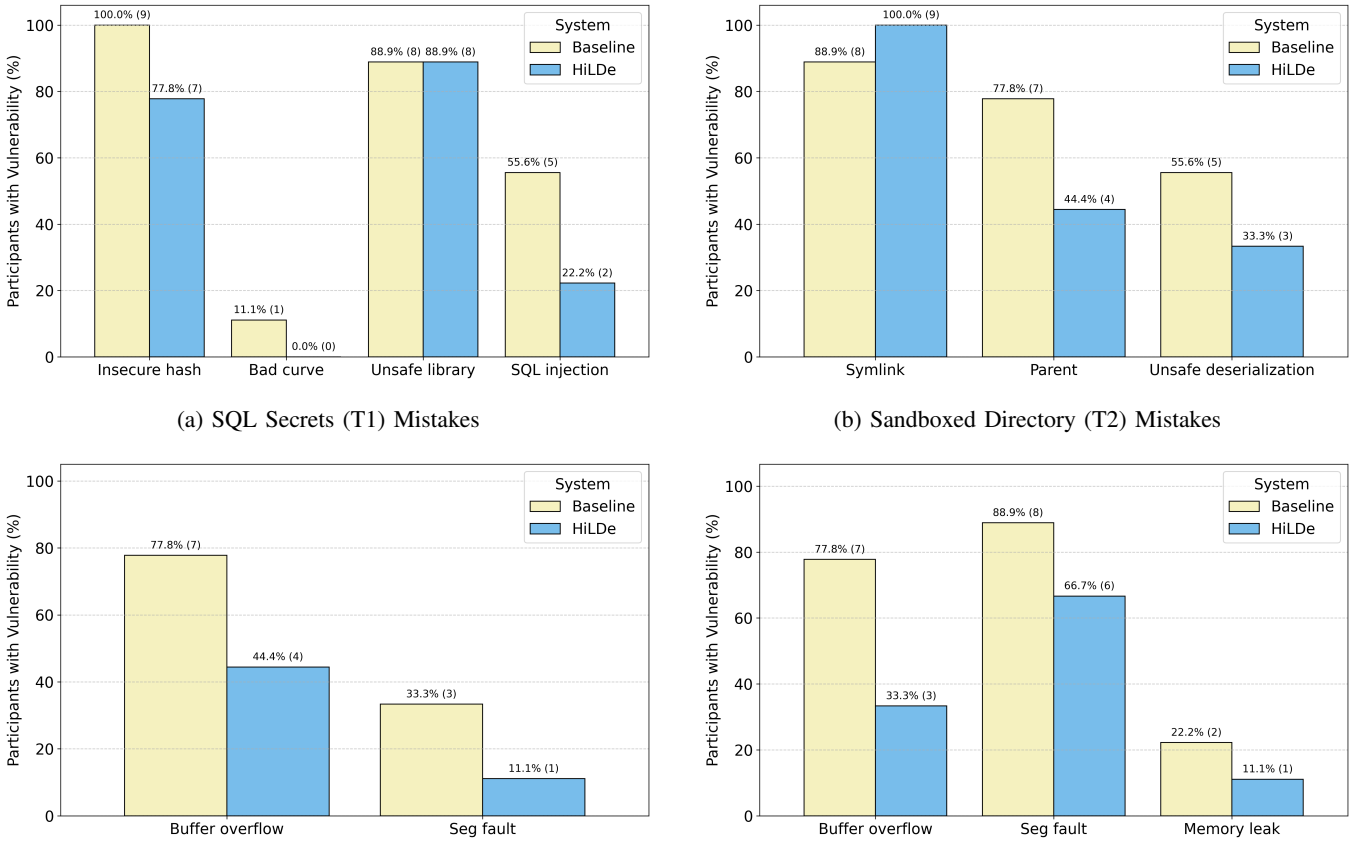
Participants were first given a brief introduction and told they would be completing programming tasks with two different AI programming assistants—`Assistant-1` (BASELINE) and `Assistant-2` (HILDE) to minimize bias. Participants were not explicitly told to focus on security; instead, they were asked to solve the tasks as instructed, and to submit code they would feel comfortable committing to a public repository to simulate a sense of personal responsibility in their code.

To account for possible order and learning effects, we split the participant pool into two groups: the "BASELINE-first" group completed the tasks using a baseline assistant, and then HILDE; the "HILDE-first" group completed the tasks using HILDE, then the baseline. We randomly assigned participants to groups while evenly distributing across task order. Both groups ended up with 9 participants.

The "HILDE-first" group was then given a walkthrough tutorial of HILDE and its features (10 minutes), after which they were asked to complete two programming tasks (15 minutes each) and two post-task surveys. Then, they repeated the same process (tutorial, two tasks, two surveys) with the BASELINE. The "BASELINE-first" group completed the same procedure, but in reverse order. Finally, all participants completed a post-study survey and a semi-structured interview (10 minutes). In total, each study session lasted at most 90 minutes. We additionally allowed participants access to a web browser, which they could use to solve any task as long as they did not consult other AI assistants.

E. Data Collection and analysis

For *quantitative* analysis, we collected participants' self-reported ratings on six metrics in a post-task survey: confidence in their solution, control over the assistant, usefulness and understanding of alternative completions, general trust in AI-generated code, and cognitive load (measured using five NASA-TLX questions [45]).



(a) SQL Secrets (T1) Mistakes

(b) Sandboxed Directory (T2) Mistakes

(c) CSV File Write (T3) Mistakes

(d) User Input to Struct (T4) Mistakes

Fig. 4: Summary of all security vulnerabilities identified in HiLDe (blue) vs. Baseline (yellow) group solutions.

We also compiled a list of security vulnerabilities that could occur in each task, and two authors used this list to independently count the number of vulnerabilities in each participant's final solutions and reach a consensus. Additionally, we measured task duration and logged every interaction with each Assistant automatically, (i.e., every Assistant query, frequency of queries, frequency of suggestion acceptances, time between a suggestion was received and then accepted/rejected, alternative suggestions viewed or accepted, participants' final solutions, etc.) for each task. Using these logs, we noted all instances where participants intentionally repaired vulnerabilities in their code, and the strategies they used to do so. We used Wilcoxon signed-rank tests to assess all differences except for repair strategies, which we analyzed via Fisher's exact tests.

For *qualitative* analysis, we recorded and transcribed each participant's session and semi-structured interview, with participant consent. Participants were encouraged to "think aloud" while they completed each task, verbalizing their problem-solving process, initial reactions to code suggestions, what they were feeling, etc. We used thematic analysis [46], [47] to identify themes from the task and interview transcripts, with a particular emphasis on instances of intentional decision-making. Two authors individually coded participant quotes from the transcripts related to our three research questions, and then collaboratively grouped these codes into broader themes to present with our quantitative results.

V. Results

In the following sections, we present a detailed quantitative and qualitative analysis using system log data and session transcripts corresponding our three research questions. Our results highlight the effectiveness of HiLDe in helping programmers write secure code with LLMs, catch more mistakes in AI suggestions, and steer code generation to match their intent.

Participants successfully completed the task and passed all tests in 70 out of 72 instances. While functional correctness was not our primary concern, we supplied a basic set of tests for each task to support participants and ensure some level of functionality. These tests allowed participants to verify that their code compiled and view its output; all but two (P12: Task 2, P18: Task 1) were able to do so within 15 minutes.

We found no significant difference in cognitive load between HiLDe and Baseline, from responses to the NASA-TLX [45] metrics on a 5-point Likert scale. Participants reported similar levels of mental demand ($M_{\text{HiLDe}} = 2.0$, $M_{\text{Baseline}} = 1.9$, $p = 0.4$), temporal demand ($M_{\text{HiLDe}} = 1.9$, $M_{\text{Baseline}} = 1.6$, $p = 0.1$), performance ($M_{\text{HiLDe}} = 4.3$, $M_{\text{Baseline}} = 4.4$, $p = 0.8$), effort ($M_{\text{HiLDe}} = 2.1$,

$M_{\text{BASELINE}} = 2.0$, $p = 0.2$), and frustration ($M_{\text{HILDE}} = 1.7$, $M_{\text{BASELINE}} = 1.6$, $p = 0.2$).

A. RQ1: Security

To investigate whether HILDE helps participants write secure code, we compared the number of vulnerabilities present in the code written with each Assistant, as well as the number of intentional security repairs made. Our overall results are shown in Fig. 4 and Fig. 5.

*1) **Participants wrote code with significantly fewer vulnerabilities using HILDE:*** Participants using HILDE generated code with 31% fewer vulnerabilities on average compared to BASELINE ($M_{\text{HILDE}} = 2.67$,$M_{\text{BASELINE}} = 3.89$,$p = 0.01$, $r = 0.53$). These results are clearly seen in Fig. 4, where the number of vulnerabilities generated by participants using HILDE (blue) is consistently lower than those using BASELINE (yellow) across most tasks. This difference is particularly impactful since participants reported having limited secure coding experience (average self-rating of 2.1 out of 5 for software security knowledge).

Across both Assistants, the most common mistakes were: insecure choice of *hashing* algorithm (T1)[3], unsafe source of *randomness* from the `ecdsa` library (T1), and *symlink* vulnerabilities (T2). Every solution written with BASELINE used insecure password hashing practices. Session logs showed that BASELINE only suggested the `SHA256` algorithm without salting, and participants did not discover resource-intensive alternatives like `scrypt` or `pbkdf2_hmac` with salting, which were available in HILDE.

In most cases, participants using HILDE generated code with as many or fewer vulnerabilities than those using BASELINE; the symlink vulnerability was the one exception: although they often mitigated the risk of parent directory traversal by selecting the `os.path.abspath` alternative to simply `path.startswith`, none used `os.path.realpath` to get the canonical path, leaving their code vulnerable to symlink attacks. In one instance, P1 was able to correct the full path traversal vulnerability with BASELINE, but only after consulting external documentation.

*2) **Participants using HILDE intentionally corrected more vulnerabilities in AI-generated code:*** Due to LLM's non-deterministic nature, both Assistants occasionally generated code that was secure by default, requiring no participant intervention. To better understand participant engagement, we distinguished between cases in which participants simply accepted already-secure code and cases where they *intentionally* steered code generation towards a more secure alternative.

Out of 42 instances of intentional security repair, 71% occurred with HILDE while only 29% occurred with BASELINE (Fig. 5). Our analysis of session logs revealed two main strategies participants used to steer code generation: a UI-driven strategy, where participants selected a more secure alternative from the Assistant's suggestions (at the completion level for BASELINE, and the token level for HILDE); and a prompt-driven strategy, where participants explicitly prompted the Assistant to generate more secure code. HILDE participants were significantly more likely to use the UI-driven approach for security repairs, while BASELINE participants relied more on explicit prompting ($percent_{\text{HILDE}} = 91$, $percent_{\text{BASELINE}} = 62$, $p = 0.03$, $r = 0.17$).

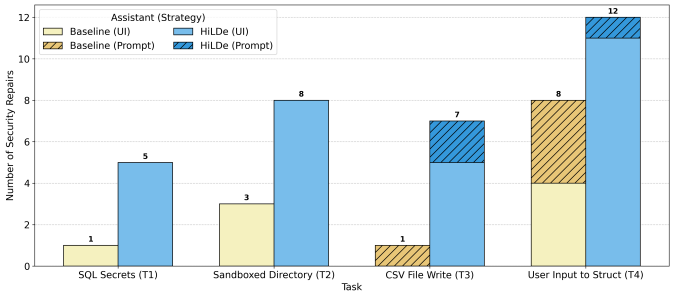


Fig. 5: Intentional security repairs using HILDE (blue) vs. BASELINE (yellow) for each repair strategy for each task.

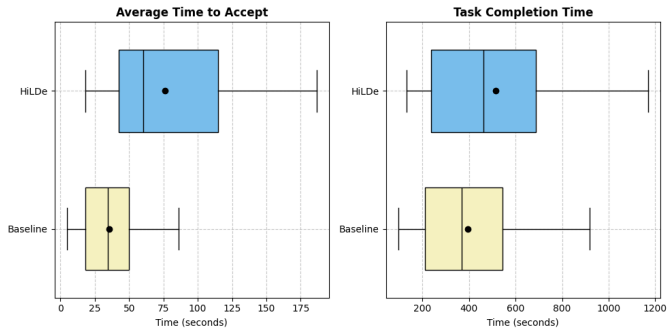


Fig. 6: Distribution for average time to accept a completion (left) and task completion time (right) for HILDE (blue) and BASELINE (yellow)

B. RQ2: Overreliance

*1) **With HILDE, participants spent more time evaluating LLM suggestions before accepting them:*** In order to understand the level of critical thinking participants demonstrated with both Assistants, we measured the average time each participant took to accept a completion after receiving it, and the average time it took participants to finish a task. We show the results in Fig. 6. Overall, we find that participants spend more time considering suggestions from HILDE before accepting them compared to BASELINE ($M_{\text{HILDE}} = 76.20, M_{\text{BASELINE}} = 35.55, t = -4.63, p < 0.001$) and they also take longer to submit their code when using HILDE ($M_{\text{HILDE}} = 514.98, M_{\text{BASELINE}} = 396.39, t = -1.97, p = 0.053$), with the latter showing a trend toward significance but being marginally above the standard threshold.

Outside quantitative data, several participants (P2, P12, P15) mentioned that HILDE encouraged them to more thoroughly evaluate the code they received from the LLM and not take it at face value. After seeing a number of hash functions suggested by HILDE, P2 said that *"if [they] had more time,*

[3]This vulnerability was illustrated in Sec. III-A.

[they] would Google each one to see which one is best", and wondered if *"there was a safer way to do it"*. P15 mentioned that "HILDE *alternatives can be useful as long as you are not blindly tabbing and accepting everything"*.

2) ***HILDE enabled users to understand the limitations of LLM-generated code****:* Participants expressed several ways in which the affordances and interaction model of HILDE allowed them to understand its shortcomings, and stopped them from blindly trusting the suggestions they get.

One common pitfall for BASELINE participants (P3, P10, P13, P16, P18) was drawing an incorrect correlation between frequency of occurrence and correctness. When seeing a particular pattern repeatedly in the global alternatives they thought *"(P3) it is standard procedure"*, *"(P13) is the default"* or *"(P18) is the only true way to do this"*.

This incorrect conclusion was less common among participants who used HILDE. A number of participants (P8, P13) seemed puzzled by the fact that the models' most likely solution was not the most correct, after seeing other alternatives. P16 asked why the model *"was not suggesting that in the first place. I am assuming most people use [unsafe alternative]...I don't know what [safe alternative] does, I know what [unsafe alternative] does cause I know the basics ...and maybe that applies to most other people, which is what is feeding the AI"*. This observation is a fair explanation for this behavior in language models.

3) ***After using* HILDE*, participants had a more accurate sense of the correctness of their solutions****:* We found no significant difference in participants' self-reported confidence in the correctness of their solutions between Assistants ($M_{\text{HILDE}} = 4.4$, $M_{\text{BASELINE}} = 4.5$, $p = 0.8$). Participants did however report having significantly higher trust in AI generated code with HILDE, though the effect size was small ($M_{\text{HILDE}} = 3.39$, $M_{\text{BASELINE}} = 3.72$, $p = 0.02$, $r = 0.23$). Participants reported high levels of confidence in general, and since those using HILDE wrote code that was more secure, their perceived correctness score was closer to their actual correctness score, compared to those using BASELINE.

Qualitatively, interactions of several participants (P12, P14, P18) with HILDE indicate a better alignment between their real and perceived performance. For instance, P12 performed the study in the BASELINE-first setting, self-reporting a high level of confidence in their solutions after the first section, while still accepting security vulnerabilities. Then, after using HILDE, they noted that it *"[gave them] more alternatives"* that were *"useful for recognizing security issues"*. They recognized that *"[they were] not familiar"* with the topic of the previous tasks and they *"likely had security issues"*, also expressing that they wanted to *"retroactively decrease [their] confidence score for* BASELINE".

C. RQ3: Achieving Programming Goals with HILDE

With* HILDE*, participants were able to achieve their programming goals more effectively There were no significant differences between Assistants in self-reported control ($M_{\text{HILDE}} = 3.8$, $M_{\text{BASELINE}} = 3.8$, $p = 0.5$), helpfulness of alternatives ($M_{\text{HILDE}} = 3.9$, $M_{\text{BASELINE}} = 4.0$, $p = 0.6$), or understanding of alternatives ($M_{\text{HILDE}} = 4.1$, $M_{\text{BASELINE}} = 3.9$, $p = 0.2$), However, our qualitative analysis of instances from Sec. V-A2 revealed that, at each LLM interaction, HILDE guided participants through a three-step process:

1) **Discover** alternative implementations that shape aspects of security, correctness, personal code style, etc.
2) **Interpret** the alternatives using contextual explanations, requirements, and preferences to understand how each implementation aligns with their intent.
3) **Act** on this understanding by selecting the most suitable implementation, effectively steering code generation.

We illustrate this pattern through two in-depth case studies focusing on participants who had different types of intent. Each case study details a participant's intentional interaction with the LLM, supplemented by similar experiences and a contrasting experience from the BASELINE group.

1) ***Case Study 1—No explicit intent****:* P15 started solving Task 4 by requesting a completion from HILDE, and immediately noticed uncertainty highlighting on the `scanf` token: *"I know with C, if you don't know what you're doing you can create some insecure stuff. So I'm assuming the LLM is just like asking if I want to use some more safe options."* (**Discover**). P15 then opened the local alternatives for `scanf`, reading the explanation for the first alternative which used the `fgets` method: "`fgets` *improves the safety by limiting the number of characters read and like yeah, buffer overflows and stuff."* (**Interpret**). After consulting and ruling out the other—irrelevant and insecure—options, P18 selected the `fgets` alternative: *"Yeah, let's go with this."* and inspected how the completion changed (**Act**).

When asked to reflect on this interaction in the post-study interview, P15 said: *"It* [the model] *wasn't sure of which function to use for user input and that's an important decision. It did make me aware of, yeah, we don't want to just accept arbitrary input."* P8 had a similar experience: *"I don't really think about the security of my code. But looking at the options, I was sort of motivated to pick something that would be more secure."* P13 echoed, "HILDE *helped me better realize my intent instead of express my intent. It helped me realize my intent because I wasn't aware of it before."*

After completing the same task with BASELINE, P10 reflected: *"I would say the only potential problem is I think I've heard that* `scanf` *is not secure or something, I don't know, maybe. But I guess for the purposes of this task, it didn't really matter."* Although P10 reviewed BASELINE's alternatives for reading user input——including some secure options——they ultimately accepted an insecure completion.

This response underscores how, without contextual affordances, P10 missed the opportunity to discover a safer alternative. In contrast, P15, P8 and P13 also started out passive in their LLM interactions, but HILDE encouraged them to discover and solidify their own programming goals.

2) ***Case Study 2—Well-defined intent****:* During the course of solving Task 2, P12 noticed that the test cases were failing because they were printing an error message instead

of raising an exception. They went ahead and tried to prompt the model via a comment to `# raise exception instead of print`. When they requested a completion from HILDE at that location, the model again suggested an unwanted print statement. However, P12 noticed the highlighting in the first token of the line, and *"was curious if [they would] find"* an appropriate solution in the alternatives (**Discover**). HILDE in fact offered an exception clause as the first option. They recognized this instantly (**Interpret**) and promptly accepted this alternative: *"Great! here we go..."* (**Act**), and verified their solution passed the tests.

In the post-task interview they remarked, *"having more alternatives show up was really nice, because I could immediately see the raise exception thing, whereas previously I might have had to re-prompt it a few times"*. They found HILDE particularly helpful since *"they [knew] the issue, and* HILDE *was just showing how to solve it"*. A number of participants (P3, P2, P10, P13) had similar observations, assuring that they appreciated *"(P3) [having] more granular control over code"*, being able to *"(P13) make smaller changes"* and *"(P2) line by line modifications"*.

On the other hand, participants had trouble accurately steering the BASELINE to make fine-grained changes. For instance, P11 found themselves in a similar situation as P12, where they knew the correct function to use in their context, and prompted the BASELINE for it through a comment. The BASELINE however *"did not suggest at all"* their preferred implementation; they had *"difficulty getting* [BASELINE]*to use it"* and had to type it out themselves. Other participants (P1, P8, P13, P16) also demonstrated this pattern of prompting via comments and not obtaining any acceptable suggestion.

Additionally, users (P1, P2, P3) complained about the inability to perform targeted changes with BASELINE, and only being able to *"(P2) change the whole structure"*. P3 compares both tools as follows: *"In* [BASELINE] *if I am not satisfied with one of the middle steps, I have to delete all the code below it and prompt [again]. But* HILDE *... it will just generate code based on the choice I made."*.

These interactions clearly indicate the advantage of HILDE when users engage with it with a clear goal in mind, as HILDE allows them to more precisely indicate their preferences at a granular level, without the need to re-prompt or regenerate, like they would need to do with BASELINE. In the case of HILDE, users have more agency in low-level LLM decision-making and are able to align it with their personal goals.

D. Recurrent BASELINE Limitations

BASELINE tended to generate alternatives that were not sufficiently diverse, and many participants (P2, P6, P11, P14, P18) found the alternatives **less helpful for discovering different implementations**. During a post-task survey, P2 noted, *"[Viewing alternatives somewhat helped because] the logic was the same, only in different format, like single line of multiple if-then-else, but the logic was the same."* P18 echoed this sentiment: *"I don't know if* [the BASELINE alternatives] *helped that much. I picked the 1st one for almost all of them if they were available. I feel like there potentially could be alternates out there for sure."*

When more diverse alternatives were accessible, participants (P6, P13, P16, P18) found that BASELINE was **less helpful for interpreting differences** between alternatives. P16 was in the "HILDE-first" group, and when they used BASELINE, they immediately expressed a desire for HILDE's local explanations: *"I don't have...the comments that tell me what's the difference between the old one and the new one. So it makes it harder to understand,* continuing, *"it takes me some time to actually see what has changed."*

Once participants understood the differences, many (P6, P8, P10, P13, P14, P16, P18) felt that BASELINE was **less helpful for deciding which alternative to accept**. Participants often accepted insecure completions from more secure alternatives because they did not have enough context. During a task P14 reflected, *"the explanations would have at least given me idea. With* [BASELINE] *I was kind of left in the dark."* Even though P18 ultimately chose a safer alternative, they reflected: *"I still don't know what* `safe_load` *means unless I go and open the docs. I didn't really know the trade-offs even though options were presented to me"*.

VI. DISCUSSION AND FUTURE WORK

A. The Benefit of Human-in-the-Loop Decoding

Our results show that Human-in-the-Loop Decoding helps programmers explore and understand a rich space of implementation choices in LLM-generated code, unlike BASELINE.

Sec. V-D finds that participants became frustrated when BASELINE generated many similar alternatives. In standard decoding algorithms, models offer little variation when they are confident in their predictions. Even if a wider variety of options were provided, it would be difficult to express every interesting choice within five global alternatives. However, adding more global alternatives is impractical from a usability standpoint——Sec. V-D finds that participants already struggled to spot meaningful differences within the small set of BASELINE alternatives, a challenge also reported by users of mainstream AI assistants [3].

In contrast, Human-in-the-Loop Decoding exposes less probable, but potentially valuable local alternatives, enabling users to discover a *broad range* of options in a *fine-grained* way. This approach aligns with general calls for AI-resilient interfaces that help users understand the range of solutions an LLM can generate from a single prompt, forming more accurate mental models of LLM behavior [4], [12].

B. Human-in-the-Loop Decoding Beyond Security

We found that participants wrote significantly safer code using HILDE, even though they were never explicitly instructed to prioritize security (Sec. V-A). However, HILDE also helped participants achieve programming goals beyond security.

For example, during Task 1, P18 noticed a SQL server `cursor` declaration highlighted in red and opened the local alternatives view. After seeing the second alternative, they declared, *"Oh yeah, usually I actually use this...the more*

idiomatic way is to use `with` *so that it automatically closes for you."* P18 selected this alternative, confirmed that the downstream output matched their intuition, and then continued solving the task. In this case, HILDE helped P18 recognize their personal coding practices in the alternatives and steer the completion to match their preferences.

While the current implementation targets security, this approach can be easily adapted for other priorities such as efficiency, maintainability, energy conservation, or personal coding style; The underlying prompt for HILDE is slightly tuned to surface security-related decision points, but adjusting the prompt could make HILDE more general or target other code attributes. For example, P9 saw the utility of HILDE in different contexts, *"whether you want to make your code secure, whether you want to make your code readable"*. We are excited to explore a more general, customizable version of HILDE in future work.

C. Intentionality vs. Efficiency in Human-AI Collaboration

Our study highlights a key trade-off: while HILDE slowed down task completion (Sec. V-B1), this additional time facilitated more intentional LLM code review and decision-making——which was often at odds with participants' usual preference for speed. For example, P10 appreciated HILDE for complicated tasks where they have to "*make multiple choices*", yet found it "*more cumbersome*" for simple code. Others (P3, P2, P13, P14, P16, P18) saw the utility in HILDE for unfamiliar domains, but preferred BASELINE when they "*(P2) want answers fast*". This distinction parallels the "exploration" and "acceleration" modes that Barke et al. [3] identified among users of AI programming assistants.

In prior user studies, task speed has often served as a measure of AI assistant utility [48]–[51], but our findings highlight how interface design can encourage more intentional code generation and help programmers strike a balance between speed and code quality.

D. HILDE Limitations

During study sessions, HILDE sometimes made unanticipated changes to downstream code after participants selected a local alternative. In a post-task survey, P11 reflected *"It was helpful to be able to change options, although, when it regenerated it just removed some of the things."* Most participants re-applied their previous selections, but some found this to be too much work, choosing to leave the completion as is.

The problem is that when a user selects a local alternative, HILDE prompts the underlying model to generate a new completion from that point, which can overwrite prior downstream edits. To mitigate this (c.f. Sec. III-B) HILDE requests several completions from the model and chooses the one most similar to the original code. However, if the user selects a token with very low-likelihood of being chosen by the model, it is unlikely that any of the generated completions will exactly match their previous edit, resulting in that change being lost. Ensuring that AI assistants preserve incremental user modifications remains an open challenge for future work [33].

Additionally, a few participants (P4, P8, P15) found HILDE's interaction method and token highlighting overwhelming at times, especially when it surfaced local alternatives that were not useful to them. As P4 remarked, *"That amount of cognitive overhead ended up...hindering my own thought process...because there were more possibilities to consider that I didn't need to consider"*. However, we found no statistically significant difference in cognitive load between the two assistants.

E. Performance Challenges

From the implementation perspective, the computational cost of the generating an initial completion and subsequent alternatives is usually high: as detailed in Sec. III-B, each one of these operations can entail hundreds of LLM calls. At the moment, this results in HILDE's latency being an order of magnitude higher than a traditional completions assistant. We believe this issue could be readily addressed by 1) effectively pruning the generation steps for which alternative previews and explanations are requested, and 2) leveraging better caching, batching and request parallelization in the generation pipeline. We leave these two proposals as directions for future work.

VII. CONCLUSION

In this paper, we present *Human-in-the-Loop Decoding*, a novel interaction technique that encourages programmers to directly influence LLM decision-making during code generation. Our implementation, HILDE, highlights critical decision points and provides contextual explanations for alternative implementations at each point. Through a security-focused user study with 18 programmers, we found that participants had more *intentional* interactions with HILDE compared to a baseline, ultimately adopting significantly safer code practices. Additionally, HILDE guided participants to achieve their programming goals—regardless of their initial intent—as they discovered diverse alternatives, interpreted how each aligned with their goals, and selected the best option to effectively steer code generation. Our findings offer valuable insights for designing LLM-driven programming interfaces that encourage more intentional, interactive code generation with end-users.

ACKNOWLEDGEMENTS

This work was supported in part by the NSF under Grant No. CCF-2107397, and Google's Gemma Academic Program GCP Credit Award. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-2038238. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring entities.

REFERENCES

[1] R. Khojah, M. Mohamad, P. Leitner, and F. G. de Oliveira Neto, "Beyond code generation: An observational study of chatgpt usage in software engineering practice," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1819–1840, 2024.

[2] J. T. Liang, C. Yang, and B. A. Myers, "A large-scale survey on the usability of ai programming assistants: Successes and challenges," in *Proceedings of the 46th IEEE/ACM international conference on software engineering*, 2024, pp. 1–13.

[3] 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.

[4] K. I. Gero, C. Swoopes, Z. Gu, J. K. Kummerfeld, and E. L. Glassman, "Supporting sensemaking of large language model outputs at scale," in *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–21.

[5] 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.

[6] 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," *Communications of the ACM*, vol. 68, no. 2, pp. 96–105, 2025.

[7] S. Oh, K. Lee, S. Park, D. Kim, and H. Kim, "Poisoned chatgpt finds work for idle hands: Exploring developers' coding practices with insecure suggestions from poisoned ai models," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 1141–1159.

[8] D. Cotroneo, C. Improta, P. Liguori, and R. Natella, "Vulnerabilities in ai code generators: Exploring targeted data poisoning attacks," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 2024, pp. 280–292.

[9] J. He and M. Vechev, "Large language models for code: Security hardening and adversarial testing," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1865–1879.

[10] A. Mohsin, H. Janicke, A. Wood, I. H. Sarker, L. Maglaras, and N. Janjua, "Can we trust large language models generated code? a framework for in-context learning, security patterns, and code evaluations across diverse llms," *arXiv preprint arXiv:2406.12513*, 2024.

[11] E. L. Glassman, Z. Gu, and J. K. Kummerfeld, "Ai-resilient interfaces," *arXiv preprint arXiv:2405.08447*, 2024.

[12] Z. Gu, I. Arawjo, K. Li, J. K. Kummerfeld, and E. L. Glassman, "An ai-resilient text rendering technique for reading and skimming documents," in *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–22.

[13] K. Ferdowsi, R. Huang, M. B. James, N. Polikarpova, and S. Lerner, "Validating ai-generated code with live programming," in *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–8.

[14] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," *arXiv preprint arXiv:1508.07909*, 2015.

[15] T. Kudo and J. Richardson, "Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," *arXiv preprint arXiv:1808.06226*, 2018.

[16] 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.

[17] H. Vasconcelos, G. Bansal, A. Fourney, Q. V. Liao, and J. W. Vaughan, "Generation probabilities are not enough: Uncertainty highlighting in ai code completions," *ACM Transactions on Computer-Human Interaction*, 2024.

[18] Y. Zhang, Q. V. Liao, and R. K. Bellamy, "Effect of confidence and explanation on accuracy and trust calibration in ai-assisted decision making," in *Proceedings of the 2020 conference on fairness, accountability, and transparency*, 2020, pp. 295–305.

[19] A. M. McNutt, C. Wang, R. A. Deline, and S. M. Drucker, "On the design of ai-powered code assistants for notebooks," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–16.

[20] A. Sajadi, B. Le, A. Nguyen, K. Damevski, and P. Chatterjee, "Do llms consider security? an empirical study on responses to programming questions," *Empirical Software Engineering*, vol. 30, no. 3, p. 101, 2025.

[21] M. Basharat and M. Omar, "Secuguard: Leveraging pattern-exploiting training in language models for advanced software vulnerability detection," *International Journal of Mathematics and Computer in Engineering*, vol. 3, no. 1, pp. 47–56, 2024.

[22] A. Shestov, R. Levichev, R. Mussabayev, E. Maslov, P. Zadorozhny, A. Cheshkov, R. Mussabayev, A. Toleu, G. Tolegen, and A. Krassovitskiy, "Finetuning large language models for vulnerability detection," *IEEE Access*, 2025.

[23] Y. Liu, L. Gao, M. Yang, Y. Xie, P. Chen, X. Zhang, and W. Chen, "Vuldetectbench: Evaluating the deep capability of vulnerability detection with large language models," *arXiv preprint arXiv:2406.07595*, 2024.

[24] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2339–2356.

[25] L. Zhang, Q. Zou, A. Singhal, X. Sun, and P. Liu, "Evaluating large language models for real-world vulnerability repair in c/c++ code," in *Proceedings of the 10th ACM International Workshop on Security and Privacy Analytics*, 2024, pp. 49–58.

[26] T. K. Le, S. Alimadadi, and S. Y. Ko, "A study of vulnerability repair in javascript programs with large language models," in *Companion Proceedings of the ACM Web Conference 2024*, 2024, pp. 666–669.

[27] G. Sandoval, H. Pearce, T. Nys, R. Karri, B. Dolan-Gavitt, and S. Garg, "Security implications of large language model code assistants: A user study," *arXiv preprint arXiv:2208.09727*, 2022.

[28] R. Yen, J. S. Zhu, S. Suh, H. Xia, and J. Zhao, "Coladder: Manipulating code generation via multi-level blocks," in *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, 2024, pp. 1–20.

[29] L. Xie, C. Zheng, H. Xia, H. Qu, and C. Zhu-Tian, "Waitgpt: Monitoring and steering conversational llm agent in data analysis with on-the-fly code visualization," in *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, 2024, pp. 1–14.

[30] Q. Guo, X. Xie, S. Liu, M. Hu, X. Li, and L. Bu, "Intention is all you need: Refining your code from your intention," *arXiv preprint arXiv:2502.08172*, 2025.

[31] P. Vaithilingam, E. L. Glassman, J. P. Inala, and C. Wang, "Dynavis: Dynamically synthesized ui widgets for visualization editing," in *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–17.

[32] L. F. Gomes, V. J. Hellendoorn, J. Aldrich, and R. Abreu, "An exploratory study of ml sketches and visual code assistants," *arXiv preprint arXiv:2412.13386*, 2024.

[33] M. Kazemitabaar, J. Williams, I. Drosos, T. Grossman, A. Z. Henley, C. Negreanu, and A. Sarkar, "Improving steering and verification in ai-assisted data analysis with interactive task decomposition," in *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, 2024, pp. 1–19.

[34] J. Zamfirescu-Pereira, E. Jun, M. Terry, Q. Yang, and B. Hartmann, "Beyond code generation: Llm-supported exploration of the program design space," in *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, 2025, pp. 1–17.

[35] S. Suh, M. Chen, B. Min, T. J.-J. Li, and H. Xia, "Luminate: Structured generation and exploration of design space with large language models for human-ai co-creation," in *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, ser. CHI '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3613904.3642400

[36] L. Yan, A. Hwang, Z. Wu, and A. Head, "Ivie: Lightweight anchored explanations of just-generated code," in *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–15.

[37] R. Cheng, T. Barik, A. Leung, F. Hohman, and J. Nichols, "Biscuit: Scaffolding llm-generated code with ephemeral uis in computational notebooks," in *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2024, pp. 13–23.

[38] J. Sun, Q. V. Liao, M. Muller, M. Agarwal, S. Houde, K. Talamadupula, and J. D. Weisz, "Investigating explainability of generative ai for code through scenario-based design," in *Proceedings of the 27th International Conference on Intelligent User Interfaces*, 2022, pp. 212–228.

[39] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Chi conference on human factors in computing systems extended abstracts*, 2022, pp. 1–7.

[40] S. S. Y. Kim, Q. V. Liao, M. Vorvoreanu, S. Ballard, and J. W. Vaughan, ""i'm not sure, but...": Examining the impact of large language models' uncertainty expression on user reliance and trust," 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. 822–835. [Online]. Available: https://doi.org/10.1145/3630106.3658941

[41] V. I. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady*, vol. 10, p. 707, Feb. 1966.

[42] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Dang *et al.*, "Qwen2. 5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.

[43] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[44] GitHub, "Github codespaces," 2024. [Online]. Available: https://github.com/features/codespaces

[45] S. G. Hart and L. E. Staveland, "Development of nasa-tlx (task load index): Results of empirical and theoretical research," in *Advances in psychology*. Elsevier, 1988, vol. 52, pp. 139–183.

[46] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative research in psychology*, vol. 3, no. 2, pp. 77–101, 2006.

[47] M. Vaismoradi, H. Turunen, and T. Bondas, "Content analysis and thematic analysis: Implications for conducting a qualitative descriptive study," *Nursing & health sciences*, vol. 15, no. 3, pp. 398–405, 2013.

[48] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The impact of ai on developer productivity: Evidence from github copilot," *arXiv preprint arXiv:2302.06590*, 2023.

[49] S. K. Kuttal, B. Ong, K. Kwasny, and P. Robe, "Trade-offs for substituting a human with an agent in a pair programming context: the good, the bad, and the ugly," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–20.

[50] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Measuring github copilot's impact on productivity," *Communications of the ACM*, vol. 67, no. 3, pp. 54–63, 2024.

[51] K. Pu, D. Lazaro, I. Arawjo, H. Xia, Z. Xiao, T. Grossman, and Y. Chen, "Assistance or disruption? exploring and evaluating the design and trade-offs of proactive ai programming support," *arXiv preprint arXiv:2502.18658*, 2025.