The RealHumanEval: Evaluating Large Language Models' Abilities to Support Programmers

Hussein Mozannar[1][2]* **Valerie Chen**[3]* **Mohammed Alsobay**[1] **Subhro Das**[2][4]

Sebastian Zhao[5] **Dennis Wei**[2][4] **Manish Nagireddy**[2][4] **Prasanna Sattigeri**[2][4]

Ameet Talwalkar[3] **David Sontag**[1][2]

[1] Massachusetts Institute of Technology Lab [2] MIT-IBM Watson AI
[3] Carnegie Mellon University [4] IBM Research [5] UC Berkeley

Abstract

Evaluation of large language models for code has primarily relied on static benchmarks, including HumanEval [10], or more recently using human preferences of LLM responses. As LLMs are increasingly used as programmer assistants, we study whether gains on existing benchmarks or more preferred LLM responses translate to programmer productivity when coding with LLMs, including time spent coding. We introduce `RealHumanEval`, a web interface to measure the ability of LLMs to assist programmers, through either autocomplete or chat support. We conducted a user study (N=213) using `RealHumanEval` in which users interacted with six LLMs of varying base model performance. Despite static benchmarks not incorporating humans-in-the-loop, we find that improvements in benchmark performance lead to increased programmer productivity; however gaps in benchmark versus human performance are not proportional—a trend that holds across both forms of LLM support. In contrast, we find that programmer preferences do not correlate with their actual performance, motivating the need for better proxy signals. We open-source `RealHumanEval` to enable human-centric evaluation of new models and the study data to facilitate efforts to improve code models.

1 Introduction

Coding benchmarks such as HumanEval [10] and MBPP [3] play a key role in evaluating the capabilities of large language models (LLMs) as programming becomes a valuable application through products such as GitHub Copilot [19] and ChatGPT [41]. These benchmarks quantify LLM abilities by measuring how well a model can complete entire coding tasks. As LLMs are increasingly adopted as programmer assistants—providing chat responses or autocomplete suggestions, rather than full code generations—prior works have argued for bringing humans-in-the-loop to evaluate LLMs [31, 11]. A predominant human-centric approach collects human preference judgments of intermediate LLM outputs, whether between pairs of LLM responses (e.g., Chatbot Arena [11]) or, for coding in particular, using programmer acceptance rates of LLM suggestions (e.g., in products

---

[*] Equal contribution.

Submitted to the 38th Conference on Neural Information Processing Systems (NeurIPS 2024) Track on Datasets and Benchmarks. Do not distribute.

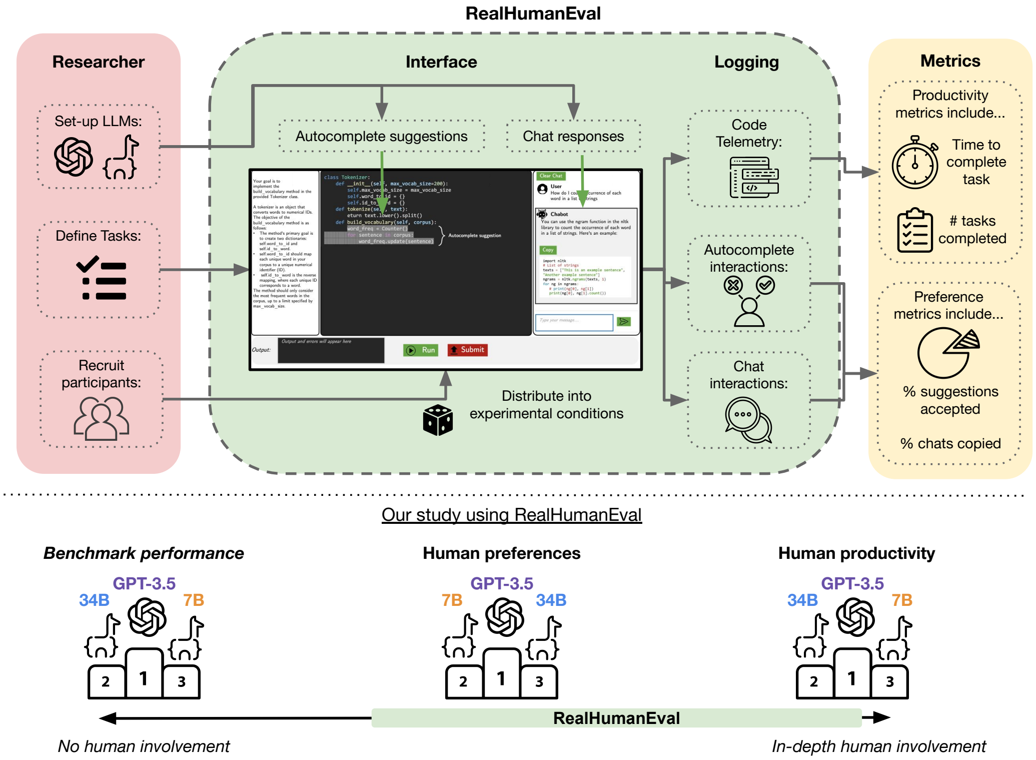


Figure 1: We introduce `RealHumanEval`, an end-to-end online evaluation platform of LLM-assisted coding through autocomplete suggestions and chat support. The goal of `RealHumanEval` is to facilitate human-centric evaluation of code LLMs, simplifying the workflow for researchers to conduct user studies to measure the effect of LLM assistance on downstream human productivity and preferences. We selected 3 families of LLMs of varying sizes (`GPT-3.5`, `CodeLlama-34b`, `CodeLlama-7b`) use `RealHumanEval` to study whether static benchmark performance or programmer preference judgments are aligned with programmer productivity.

such as Github Copilot [5]). However, such evaluation may not capture the LLM's downstream impact on programmer productivity.

Evaluating the utility of LLMs on downstream productivity requires conducting user studies where programmers code with LLM assistance. While a set of small-scale user studies have been conducted to primarily build a qualitative understanding of how programmers use LLM assistance, they are typically restricted to evaluations on one model and one form of LLM support, primarily relying on commercial tools like Github Copilot or ChatGPT [4, 36, 51, 47, 32, 43]. To enable evaluations of a broader set of LLMs and lower the barrier to conducting these studies, we introduce an online evaluation platform, `RealHumanEval` (Figure 1). The platform consists of a code editor where programmers can solve coding tasks with two common forms of LLM assistance: programmers can either ask questions to the LLM through a chat window or receive code completion suggestions through an autocomplete system inside the editor. The interface also supports executing and testing code and logging telemetry which can be used to compute productivity metrics, including time to complete a task or number of tasks completed, and preference metrics, including average acceptance rates of suggestions and the likelihood of copying code from chat responses.

Using `RealHumanEval`, we conduct a user study with 213 participants to understand the effect of a model's benchmark performance and the form of LLM assistance on downstream productivity metrics. Each participant was assigned to one of seven conditions: a control condition with no LLM support, three conditions with autocomplete support from either `CodeLlama-7b` [48], `CodeLlama-34b` [48], or `GPT-3.5-turbo-instruct`[7], and finally three conditions where the editor is equipped with a chat window powered by the chat variants of the previous models. We deliberately select three model families with increasingly higher benchmark performance and consider model pairs within each family with similar benchmark performance to understand the effect of autocomplete versus chat assistance. Through the study, we collected a dataset of interactions on 771 coding total tasks, where 5204 autocomplete suggestions were shown and 775 chat messages were sent.

Overall, we find that improving a model's base performance on existing coding benchmarks leads to gains in human productivity, particularly in the time spent completing tasks. These trends were present across both chat and autocomplete interactions, validating the potential "generalizability" of benchmarks to more realistic contexts. However, we observe that gaps in benchmark versus human performance are not necessarily proportional, suggesting that further gains in benchmark performance do not necessarily translate into equivalent gains in human productivity. We also investigated whether human preference metrics, such as the average acceptance rate of suggestions and the likelihood of copying code from chat responses, are aligned with productivity metrics. While these preference metrics are readily available in real deployments of LLM systems compared to task completion time and thus can be attractive proxy metrics [60], we find that they are only correlated with programmer perceptions of LLM helpfulness but not necessarily with actual programmer performance. The dissimilar findings between benchmarking and human preference metrics highlight the importance of careful evaluation to disentangle which metrics are indicative of downstream performance.

In summary, our contributions are as follows: (1) an open-source platform `RealHumanEval` to encourage more human-centric evaluations of code LLMs, (2) an evaluation of 6 code LLMs of varying performance using `RealHumanEval` to provide insights into the alignment and discrepancies between benchmark performance and human preferences with downstream user productivity. Our findings emphasize the importance of studying how programmers interact with code LLMs through user studies to identify nuances in programmer-LLM interactions. Finally, (3) we release the dataset of interactions collected from this study to guide the development of better coding assistants.[2]

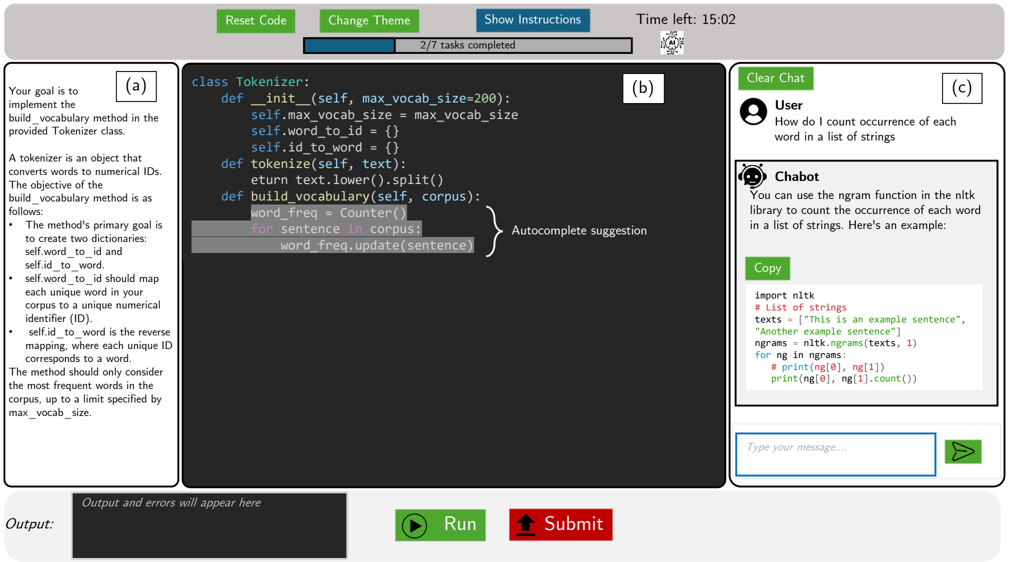
2 Related Work

Coding Benchmarks. Benchmarks are essential for tracking the progress of LLMs, and coding benchmarks are a key piece [1, 29, 57, 21]. Moreover, the coding ability of an LLM can be informative of its reasoning abilities [35]; thus, performance on coding benchmark is of broader interest. While HumanEval [10] and MBPP [3] are the most commonly used coding benchmarks, many extensions and further benchmarks have been proposed [34, 40, 59, 33, 23, 26, 56, 55], we highlight a few: EvalPlus extends HumanEval's test cases [33], MultiPL-E [9] to other languages, ReCode with robustness checks [53], HUMANEVALPACK [38] with code repair and explanation tasks, and buggy-HumanEval [17] with bugs in the reference code. Relatedly, the DS-1000 [28] benchmark evaluates models' abilities on data science problems that require using external libraries. More involved evaluations include the multi-turn program evaluation benchmark [40] and SWE-bench [23], which requires the LLM to resolve GitHub issues. While existing benchmarks evaluate a diverse set of LLM behaviors across models, these benchmarks do not, however, include a programmer-in-the-loop, as there would be in a real-world setup. Our evaluation complements this existing line of work by conducting a user study, where programmers put these behaviors to the test in realistic scenarios.

Preference Metrics. Instead of relying solely on coding benchmarks' `pass@k` metrics, which consider only the functional correctness of LLM-generated code, recent work has advocated for incorporating human preferences, which may better reflect how LLM code could be useful to a programmer without necessarily being functionally correct [16]. Preferences are generally collected after a single turn (e.g., after a single LLM response or suggestion) and thus can be collected at scale [5, 11] or even simulated with LLMs [18, 58]. Given that preferences are only a form of intermediate feedback, in this study, we evaluate whether human preferences provide a signal for downstream productivity gains when coding with LLMs.

---

[2] The code for the interface, data, and our analyses can be found at: `https://github.com/clinicalml/realhumaneval`.



Your goal is to implement the build_vocabulary method in the provided Tokenizer class.

A tokenizer is an object that converts words to numerical IDs. The objective of the build_vocabulary method is as follows:

- The method's primary goal is to create two dictionaries: self.word_to_id and self.id_to_word.
- self.word_to_id should map each unique word in your corpus to a unique numerical identifier (ID).
- self.id_to_word is the reverse mapping, where each unique ID corresponds to a word.

The method should only consider the most frequent words in the corpus, up to a limit specified by max_vocab_size.

```
class Tokenizer:
    def __init__(self, max_vocab_size=200):
        self.max_vocab_size = max_vocab_size
        self.word_to_id = {}
        self.id_to_word = {}
    def tokenize(self, text):
        eturn text.lower().split()
    def build_vocabulary(self, corpus):
        word_freq = Counter()
        for sentence in corpus:
            word_freq.update(sentence)
```

How do I count occurrence of each word in a list of strings

You can use the ngram function in the nltk library to count the occurrence of each word in a list of strings. Here's an example:

```
import nltk
# List of strings
texts = ["This is an example sentence",
"Another example sentence"]
ngrams = nltk.ngrams(texts, 1)
for ng in ngrams:
   # print(ng[0], ng[1])
   print(ng[0], ng[1].count())
```

Figure 2: We introduce `RealHumanEval`, an online evaluation platform for LLM-assisted coding. The platform consists of (a) a customizable task description, (b) the code editor which shows autocomplete suggestions in grey, and (c) the chat assistant. Above the editor, users can check their task progress and the amount of time left, reset the editor, change the editor theme, and view study instructions. Below the editor, they can run and submit their code.

Programmer-LLM Interaction. Prior work conducting user studies where programmers code with LLM assistance has primarily focused on two forms of LLM support, autocomplete suggestions [51, 43, 4, 45, 36, 52, 14] and chat dialogue [47, 12, 25, 20, 39]. While these studies have made progress in understanding programmer-LLM interactions, all studies only consider one LLM—often Copilot or ChatGPT—and one form of LLM support—either autocomplete or chat, making it difficult to compare outcomes and metrics *across models* and *across forms of support*. We contribute a web platform `RealHumanEval` to enable ease of human-centric evaluation of more models and forms of support (see Appendix A for a more in-depth comparison). Beyond applications of coding assistance, our study contributes to the broader literature studying human interactions with LLMs [31, 13, 30, 15, 22, 27, 24, 8].

3 RealHumanEval

We introduce `RealHumanEval`, a web-based platform to conduct human-centric evaluation of LLMs for programming through the workflow shown in Figure 1. We created `RealHumanEval` to facilitate large-scale studies of programmers coding with LLMs, eliminating the need for participants to perform any additional installation of a bespoke IDE or study-specific extension or to have access to special hardware to serve study-specific models.

Interface. As shown in Figure 2, `RealHumanEval` incorporates many basic features of common code editors and the functionality of programming interview sites such as LeetCode. Given a coding task that consists of a natural language description, partial code (e.g., a function signature), and unit tests that evaluate the task, `RealHumanEval` allows the programmer to write code with assistance from an LLM to complete the task. The platform has a panel that displays the natural language description of a task, as shown in Figure 2(a), alongside partial code to solve the task. Participants then write their code for the task in the code editor and can test their code with a button that checks the code against test cases and runs their code directly. The editor displays any errors, if available, and whether the code passes the unit test. Once the programmer completes the task, a new task can be loaded into the interface. For our user study, we only use a single code editor file, however, `RealHumanEval` can support multiple-file projects.

Forms of LLM Assistance. `RealHumanEval` supports two forms of LLM assistance: *autocomplete-based* and *chat-based*. Examples of autocomplete and chat assistants include GitHub's Copilot [19], Replit's Ghostwriter [46], Amazon CodeWhisperer [2], and ChatGPT [41]. In *autocomplete-based* assistance, the programmer writes code in an editor, and the LLM displays a code suggestion inline, which is greyed out as shown in Figure 2(b). The LLM is assumed to be able to fill in code given a suffix and prefix. A suggestion, based on the current code body in the editor, appears whenever the programmer pauses typing for more than two seconds or when the programmer requests a suggestion by pressing a hotkey. The programmer can accept the suggestion by pressing the tab key or reject it by pressing escape or continuing to type.

In *chat-based* assistance, the programmer writes code in an editor and has access to a side chat window where the programmer can ask questions and get responses from the LLM, as illustrated in Figure 2(c). The LLM is assumed to be a chat model. The programmer can copy and paste code from the LLM's responses into the editor. Currently, the interface supports any LLM invoked via an online API. Further information on the implementation of both forms of assistance is in Appendix B.

Telemetry logging. `RealHumanEval` logs all user behavior, including interactions with LLM support. For each autocomplete suggestion, we log the following tuple $\{(P_i, S_i), R_i, A_i\}_{i=1}^{n}$ where (P_i, S_i) is the prefix and suffix of the code based on cursor position at the time of suggestion i, R_i is the LLM suggestion, and A_i is a binary variable indicating whether the suggestion was accepted. All the logs are stored in a dataset $\mathcal{D}_{\text{ac}}$. For chat-assistance, we log for each user message the following tuple $\{X_i, M_i, R_i, C_i\}_{i=1}^{n}$ where X_i is the code at the time of message i, M_i is the user message (including prior chat history), R_i is the response from the LLM for the message, and C_i is the number of times code was copied from the LLM's response. All the logs are stored in a dataset $\mathcal{D}_{\text{chat}}$. Moreover, every 15 seconds, the interface saves the entire code the user has written.

Metrics. From the telemetry logs, `RealHumanEval` provides multiple metrics to analyze programmer behaviors: the *number of tasks completed* (completion is measured by whether the submitted code passes a set of private test cases), *time to task success* (measured in seconds), *acceptance rate* (fraction of suggestions shown that are accepted, for autocomplete), and *number of chat code copies* (counting when user copies code from LLM response, for chat) among other metrics.

4 Study Design

Using `RealHumanEval`, we conducted a user study to evaluate (1) the impact of LLM assistance on programmer performance as a function of the LLM's performance on static benchmarks and (2) whether human preference metrics correlate with programmer productivity metrics.

Overview. For the entire duration of the study, participants are randomly assigned either to a control group, where they experienced the `no LLM` condition, or to the LLM-assisted group, where they experienced the *autocomplete* or *chat support* condition. For autocomplete-based support, the window in Figure 2(c) is hidden. For chat-based support, no autocomplete suggestions are shown in Figure 2(b). Participants are only assigned to one condition to minimize context switching, given the relatively short duration of the study. The study was conducted asynchronously using the `RealHumanEval` platform; participants were told not to use any outside resources (e.g., Google), and cannot paste any text originating outside the app into the editor. Specific instructions are in Appendix B. The first problem was a simple task (i.e., compute the sum and product of a list) for participants to familiarize themselves with the interface. Participants are given 35 minutes to complete as many tasks as possible. If 10 minutes pass and the participant has not completed the task, a button appears to provide the option to skip the task.

Tasks. We designed 17 coding tasks for the platform that can be categorized into three categories: (a) *algorithmic problems* from HumanEval (e.g., solve interview-style coding), (b) *data manipulation problems* (e.g., wrangle input dataframe into desired output), and (c) *editing and augmenting code* *tasks* (e.g., fill in provided code scaffold to achieve desired behavior). While the set of tasks does not evaluate all types of coding problems exhaustively, they do capture tasks of varying difficulty and solutions of varying length, as well as the use of different programming skills, leading to varying opportunities to benefit from LLM support. We chose 17 tasks to build diversity across tasks while being able to collect enough samples per task. We ensured that no LLM model considered in the study, in addition to GPT-4, could solve all tasks perfectly, so that programmers would not simply accept all LLM suggestions and that each task could be solved in under 20 minutes by an experienced programmer (validated through pilots with the authors and volunteer participants), to ensure that these were reasonable questions to consider for a user study. These 17 tasks are distributed into five sets, where each set consists of a different mix of task types in varying orders but shares the first two tasks. Each participant is randomly assigned to one of these sets. The LLMs are not aware of the task descriptions unless the programmer types them in the editor or chat window; this is to simulate the real world where the task description represents the programmer's hidden true intent. We provide examples of the coding tasks in Appendix C and in full in the supplementary materials.

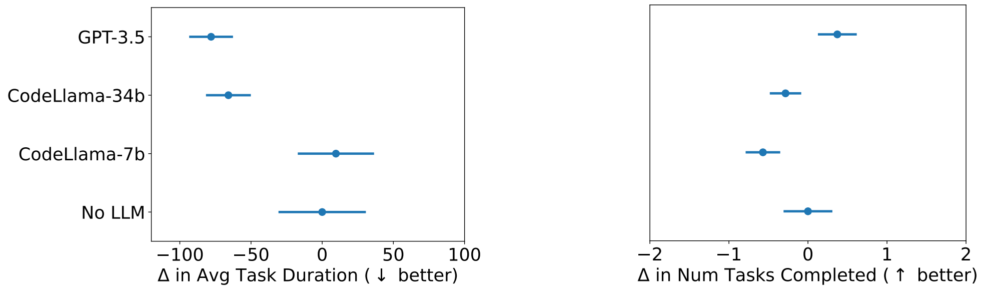
Conditions. For the autocomplete conditions, we chose base LLM models that naturally generate next-word predictions, whereas the "chatty" variants of the base models are employed for the chat conditions. To evaluate the effect of LLM capabilities, we selected three types of models that demonstrate clear gaps in performance on existing benchmarks (as shown in Figure 8). In total, we selected 6 LLMs for our study: 4 from the Code Llama family [48] (`CodeLlama-7b`, `CodeLlama-7b-instruct`, `CodeLlama-34b`, `CodeLlama-34b-instruct`), along with two models from the GPT series [7] (`GPT-3.5-turbo` and `GPT-3.5-turbo-instruct`). To avoid confusion, we refer to the autocomplete conditions by the base name of the model: `CodeLlama-7b`, `CodeLlama-34b` and `GPT-3.5` (refers to `GPT-3.5-turbo-instruct`); and the chat conditions by the base name of the model with chat: `CodeLlama-7b (chat)` (refers to `CodeLlama-7b- instruct`), `CodeLlama-34b (chat)` (refers to `CodeLlama-34b- instruct`) and `GPT-3.5 (chat)` (refers to `GPT-3.5-turbo`). Specific choices of parameters, system prompts, and other considerations are provided in Appendix D.

Participants. We recruited 229 total participants from university mailing lists and social media to capture a range of coding experiences. We verified that participants were above 18 years of age, resided in the United States, and correctly completed a simple Python screening question. Out of the 229 participants, we filtered out those who did not complete any task or did not write code for a period of 15 minutes during the study to arrive at 213 final participants. Of the 229 participants, 34% identify as Female. In terms of occupation, 79% are Undergraduate or Graduate Students studying computer science, 13% work in Software Development and 7% work in AI. While a majority of our participants were students, only 34% of participants had less than 2 years of professional programming experience. We ensured that participants were roughly equally distributed across experimental conditions based on programming experience. 11% had never used any form of AI for coding while 67% of participants use AI at least once a week for coding. Participants were provided with a $15 Amazon gift card as compensation. This study was approved by institutional IRB review.

User study metrics. To quantify the benefits of LLM assistance on the number of tasks completed and time to task success, we report the gap between each condition where some form of LLM assistance was provided and the control no LLM condition, which we denoted as Δ. For example, for time to task success, $\Delta < 0$ for LLM support indicates that participants took less time to complete tasks with the LLM. In addition to the quantitative metrics, we also ask post-study questions to obtain participants' subjective measures of their interactions with the LLM: we ask participants to rate the helpfulness of the LLM on a scale of $[1, 10]$ and to describe how the LLM support provided (if any) was helpful and how it could be improved. We also measure two preference metrics, suggestion acceptance rate and percentage of chat code copies.

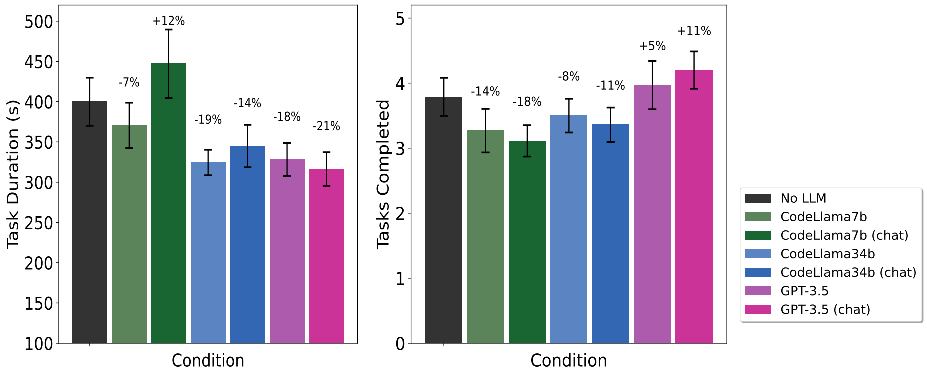
5 Results

We report results for data collected from 213 participants split across the seven conditions; since condition assignment is random, each condition has around 25 to 35 participants (except for `No LLM`,



(a) Difference in task completion time (in seconds) comparing LLMs to the No LLM condition.

(b) Difference in number of tasks completed compared to the No LLM condition.



(c) Average task completion time (in seconds) by condition.

(d) Average number of tasks completed by condition.

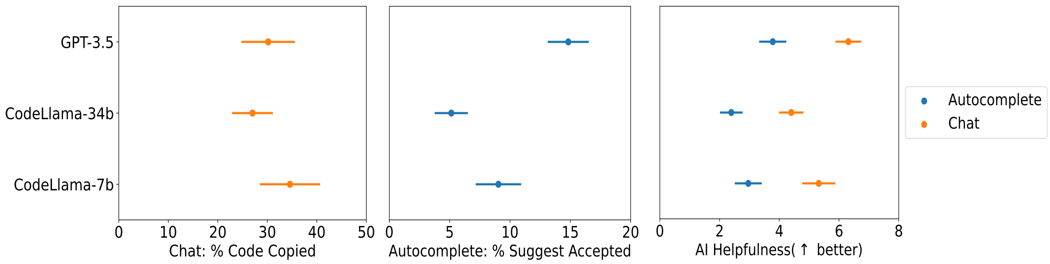
Figure 3: We measure the effect of LLM support on user study performance on mean task duration in seconds (a,c) and number of tasks completed across model type (b,d). In (a) and (b), we compute Δ, the difference between each model type—aggregating conditions corresponding to the same model type, e.g., Codellama7b and Codellama7b (chat)—and the `No LLM` condition for each metric. In (c) and (d), we break down the same metrics for each of the seven conditions and mark the percentage improvement over the `No LLM` condition. We observe that better LLM support can improve task completion time, but not necessarily increase the number of tasks completed. Error bars denote *standard errors*—the standard deviation divided by the square root of the sample size (i.e., across participants), where each participant contributes a single data point.

which has 39 participants). Participants completed a total of 771 coding tasks (mean of 3.6 tasks per person) on average in 334 seconds (std=238 seconds), were shown 5204 autocomplete suggestions ($|\mathcal{D}_{\text{ac}}|$), with an average 11.3% acceptance rate, and received 775 messages from the chat LLMs ($|\mathcal{D}_{\text{chat}}|$), with 29.6% of messages having at least one copy event. In the following analyses, we conduct ordinary least squares regressions with Benjamini-Hochberg correction and use a significance level of 0.05. A more in-depth analysis of both datasets and results is in Appendix E.

Providing LLM assistance reduces the amount of time spent coding. To measure the productivity gains of LLM assistance to programmers, we look at two metrics: the amount of time spent coding (in seconds) and the number of tasks completed. We first distill our observations for each metric by comparing performance for each model type (i.e., combining autocomplete and chat models) against the `No LLM` condition.[3] As shown in Figure 3(a), we find that compared to the `No LLM` setting where participants spent an average of 400 seconds per task, both `GPT-3.5` and `CodeLlama-34b` models reduce the amount of time spent per task by an average of 78 and 64 seconds respectively ($p = 0.04$ and $p = 0.12$). In contrast, `CodeLlama-7b` models slightly increase the average time spent on a task by 10 seconds. However, we do not observe statistical differences across *any* of the conditions in

---

[3] In Appendix E, we repeated the same analyses controlling for task difficulty and observed the same trends.



(a) Percentage of chat messages copied for chat conditions.

(b) Percentage of autocomplete suggestions accepted.

(c) Rating of LLM helpfulness across both autocomplete and chat conditions.

Figure 4: Measuring participant preferences of different models by the amount of interaction with chat (a) or autocomplete systems (b), with standard error. We find that preference judgments align with the reported helpfulness of the LLM assistant post-study (c); however, these preferences do not necessarily align with their actual task performance.

the number of tasks completed, as shown in Figure 3(b), meaning no form of LLM support allowed programmers to solve *more* problems than they otherwise would have on their own. We hypothesize that benefits in task completion were not observed because of the short duration of the user study (35 minutes) and the amount of time it takes to complete each task, though we do observe an increase in the number of tasks attempted.

We now consider how our observations using `RealHumanEval` implicate the broader code LLM evaluation landscape, specifically the use of (1) static benchmarks and (2) human preference metrics.

(1) Are LLM performance on static benchmarks informative of user productivity with LLM assistance? We find that improvements in model-specific evaluations on benchmarks also improve human performance on both productivity metrics in the user study (i.e., `CodeLlama-7b` models led to the least number of tasks completed, while `GPT-3.5` models led to the most). Interestingly, this trend holds even when considering metrics with chat and autocomplete separately, in Figure 3(c-d). *However*, significant gaps in benchmark performance result in relatively indistinguishable differences in terms of human performance. For instance, `CodeLlama-34b (chat)` is 19% better over `CodeLlama-7b (chat)` models on HumanEval, and participants are 22.8% (95% CI [2.8, 38.7]) faster on average to complete a task with 34b vs 7b. Yet, `GPT-3.5 (chat)` model outperforms `CodeLlama-34b (chat)` by 85% on HumanEval, and yet participants equipped with `GPT-3.5 (chat)` models are only 8.3% (95% CI [-11.2, 24.6]) faster than those with `CodeLlama-34b (chat)`. While we do not necessarily expect performance gaps to be consistent, this finding suggests that, after a certain point, additional gains on static benchmarks may not translate to practical utility.

(2) Do human preferences align with productivity? We also consider programmer preferences for the LLM assistant's suggestions on autocomplete and chat: the average suggestion acceptance rate and the average copies-per-response respectively. While both `GPT-3.5` and `CodeLlama-34b` models reduced the amount of time spent coding over `CodeLlama-7b`, we do not find the same trends reflected in human preferences. As shown in Figure 4(a), we find that suggestions from `CodeLlama-34b` are less likely to be accepted at 5% compared to 15% and 9% for `GPT-3.5` and `CodeLlama-7b` ($p < 0.001$ and $p = 0.19$). The same ordering occurs for the percentage of chat messages copied (27% versus 35% and 29%, though not significant) in Figure 4(b). By analyzing the participants' qualitative responses, discussed in Section F, we identify potential factors that may have contributed to these preferences, including a perceived lack of context in `CodeLlama-34b` suggestions and a slight increase in latency in `CodeLlama-34b (chat)` responses. These results suggest that various external factors that might be difficult to anticipate a priori can easily affect human preferences even if they do not impact downstream productivity.

5.1 Additional User Study Observations

Findings on the effect of the form of LLM support and task type further illustrate the importance of evaluation with humans in the loop.

Chat support is perceived to be more helpful than autocomplete support. Even though autocomplete and chat variants obtained similar performance on static benchmarks and participant performance in both conditions conditioned on a model type was relatively similar, we observe that chat models are rated by participants in the post-study questions as significantly more helpful than autocomplete models ($p < 0.001$), as shown in Figure 4(c). Again, we observe that `CodeLlama-34b` models tend to be rated as less helpful (3.3 out of 10), than the other two models (4.19 and 5.09 out of 10 for `CodeLlama-7b` and `GPT-3.5`).

The benefits of LLM assistance can vary by task type. We also analyze the time spent on each task category, comparing when participants have access to LLM assistance versus the control condition. As shown in Figure 12, we find suggestive evidence that LLM assistance was particularly effective in reducing the time programmers needed to solve data manipulation tasks, by 28.35%, and slightly less so for problems that required editing and augmenting existing code, by 13.48%. In contrast, we found that LLMs were unhelpful on algorithmic problems, increasing the amount of time spent by 11.7%. A breakdown by individual task is in Appendix E.

6 Discussion

In this work, we introduce `RealHumanEval`, a human-centric evaluation platform for code LLMs, and conduct a user study using the platform to measure programmer productivity assisted by different LLMs. We believe `RealHumanEval` can be adopted to evaluate newly released LLM models in a more meaningful way and become a standard for evaluation. To enable this, our interface is designed to be easily repurposed for future user studies and evaluations by the community and extended to evaluate new ways of interacting with LLMs for programming.

Recommendations for future work. We summarize participant suggestions on how coding assistants could be improved (more detail in Appendix F). Participants overwhelmingly felt that LLMs struggled to infer the appropriate *context* to provide the most useful support from the information available, highlighting the need for benchmarks that capture settings where LLMs need to infer intent from partial or fuzzy instructions. The suggestion also underscores the importance of evaluating LLMs with humans-in-the-loop; we recommend the community leverage and build on `RealHumanEval` to evaluate new LLMs' coding abilities. There are also opportunities to improve autocomplete and chat assistants to be better programming partners [54]. For example, autocomplete systems might benefit from personalization of when participants would benefit from suggestions and dynamically adjusting the length, while chat-based systems could be improved to have better, more tailored dialogue experience and better integration with the editor. Toward these goals, we release the datasets of user interactions that can be leveraged as signals of user preferences and behavior patterns.

Limitations. Firstly, we acknowledge that a set of 17 coding tasks does not span the entire set of tasks a professional programmer might encounter in their work and may limit the generalizability of our evaluations of the 6 models. We encourage future work to leverage `RealHumanEval` to conduct further studies with a more extensive set of tasks. Second, the coding tasks we used are of short duration, while real-world programming tasks can take hours to months. This presents a trade-off in study design: short tasks allow us to evaluate with more participants and models in a shorter period but give us a less clear signal compared to longer-term tasks. Third, `RealHumanEval` does not fully replicate all functionality existing products such as GitHub Copilot may have so the study may underestimate exact productivity benefits. Such products are complex systems comprising more than a single LLM, where many details are hidden and thus not easily replicable. We release `RealHumanEval` to enable others to build more functionality in an open-source manner.

Societal implications. While our evaluations focused on productivity metrics, there are additional metrics of interest that may be important to measure when studying programmer interactions with LLM support. On the programmer side, further evaluations are needed to understand whether programmers appropriately rely on LLM support [50] and whether LLM support leads to potential de-skilling [6]. Further, our metrics do not consider potential safety concerns, where LLMs may generate harmful or insecure code [42, 44].

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] Amazon. Ml-powered coding companion – amazon codewhisperer, 2022.

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[4] Shraddha Barke, Michael B James, and Nadia Polikarpova. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):85–111, 2023.

[5] Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. Taking flight with copilot: Early insights and opportunities of ai-powered pair-programming tools. *Queue*, 20(6):35–57, 2022.

[6] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[8] Erik Brynjolfsson, Danielle Li, and Lindsey R Raymond. Generative ai at work. Technical report, National Bureau of Economic Research, 2023.

[9] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. Multipl-e: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 2023.

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[11] Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E Gonzalez, et al. Chatbot arena: An open platform for evaluating llms by human preference. *arXiv preprint arXiv:2403.04132*, 2024.

[12] Bhavya Chopra, Ananya Singha, Anna Fariha, Sumit Gulwani, Chris Parnin, Ashish Tiwari, and Austin Z Henley. Conversational challenges in ai-powered data science: Obstacles, needs, and design opportunities. *arXiv preprint arXiv:2310.16164*, 2023.

[13] Katherine M Collins, Albert Q Jiang, Simon Frieder, Lionel Wong, Miri Zilka, Umang Bhatt, Thomas Lukasiewicz, Yuhuai Wu, Joshua B Tenenbaum, William Hart, et al. Evaluating language models for mathematics through interactions. *arXiv preprint arXiv:2306.01694*, 2023.

[14] Kevin Zheyuan Cui, Mert Demirer, Sonia Jaffe, Leon Musolff, Sida Peng, and Tobias Salz. The productivity effects of generative ai: Evidence from a field experiment with github copilot. 2024.

[15] Hai Dang, Karim Benharrak, Florian Lehmann, and Daniel Buschek. Beyond text generation: Supporting writers with continuous automatic text summaries. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, pages 1–13, 2022.

[16] Victor Dibia, Adam Fourney, Gagan Bansal, Forough Poursabzi-Sangdeh, Han Liu, and Saleema Amershi. Aligning offline metrics and human judgments of value for code generation models. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 8516–8528, 2023.

[17] Tuan Dinh, Jinman Zhao, Samson Tan, Renato Negrinho, Leonard Lausen, Sheng Zha, and George Karypis. Large language models of code fail at completing code with potential bugs. *Advances in Neural Information Processing Systems*, 36, 2023.

[18] Yann Dubois, Chen Xuechen Li, Rohan Taori, Tianyi Zhang, Ishaan Gulrajani, Jimmy Ba, Carlos Guestrin, Percy S Liang, and Tatsunori B Hashimoto. Alpacafarm: A simulation framework for methods that learn from human feedback. *Advances in Neural Information Processing Systems*, 36, 2023.

[19] Github. Github copilot - your ai pair programmer, 2022.

[20] Ken Gu, Ruoxi Shang, Tim Althoff, Chenglong Wang, and Steven M Drucker. How do analysts understand and verify ai-assisted data analyses? In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pages 1–22, 2024.

[21] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620*, 2023.

[22] Maurice Jakesch, Advait Bhat, Daniel Buschek, Lior Zalmanson, and Mor Naaman. Co-writing with opinionated language models affects users' views. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–15, 2023.

[23] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2023.

[24] Eunkyung Jo, Daniel A Epstein, Hyunhoon Jung, and Young-Ho Kim. Understanding the benefits and challenges of deploying conversational ai leveraging large language models for public health intervention. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–16, 2023.

[25] Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J Ericson, David Weintrop, and Tovi Grossman. Studying the effect of ai code generators on supporting novice learners in introductory programming. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–23, 2023.

[26] Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. *arXiv preprint arXiv:2303.03004*, 2023.

[27] Andreas Köpf, Yannic Kilcher, Dimitri von Rütte, Sotiris Anagnostidis, Zhi Rui Tam, Keith Stevens, Abdullah Barhoum, Duc Nguyen, Oliver Stanley, Richárd Nagyfi, et al. Openassistant conversations-democratizing large language model alignment. *Advances in Neural Information Processing Systems*, 36, 2023.

[28] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR, 2023.

[29] Md Tahmid Rahman Laskar, M Saiful Bari, Mizanur Rahman, Md Amran Hossen Bhuiyan, Shafiq Joty, and Jimmy Huang. A systematic study and comprehensive evaluation of chatgpt on benchmark datasets. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 431–469, 2023.

[30] Mina Lee, Percy Liang, and Qian Yang. Coauthor: Designing a human-ai collaborative writing dataset for exploring language model capabilities. In *CHI Conference on Human Factors in Computing Systems*, pages 1–19, 2022.

[31] Mina Lee, Megha Srivastava, Amelia Hardy, John Thickstun, Esin Durmus, Ashwin Paranjape, Ines Gerard-Ursin, Xiang Lisa Li, Faisal Ladhak, Frieda Rong, Rose E Wang, Minae Kwon, Joon Sung Park, Hancheng Cao, Tony Lee, Rishi Bommasani, Michael S. Bernstein, and Percy Liang. Evaluating human-language model interaction. *Transactions on Machine Learning Research*, 2023.

[32] Jenny T Liang, Carmen Badea, Christian Bird, Robert DeLine, Denae Ford, Nicole Forsgren, and Thomas Zimmermann. Can gpt-4 replicate empirical software engineering research? *arXiv preprint arXiv:2310.01727*, 2023.

[33] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36, 2023.

[34] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[35] Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. Language models of code are few-shot commonsense learners. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 1384–1403, 2022.

[36] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric 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*, pages 1–16, 2024.

[37] Hussein Mozannar, Valerie Chen, Dennis Wei, Prasanna Sattigeri, Manish Nagireddy, Subhro Das, Ameet Talwalkar, and David Sontag. Simulating iterative human-ai interaction in programming with llms. In *NeurIPS 2023 Workshop on Instruction Tuning and Instruction Following*, 2023.

[38] Niklas Muennighoff, Qian Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models. In *The Twelfth International Conference on Learning Representations*, 2023.

[39] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[40] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*, 2023.

[41] OpenAI. Chatgpt: Optimizing language models for dialogue, 2022.

[42] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE, 2022.

[43] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590*, 2023.

[44] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. Do users write more insecure code with ai assistants? In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2785–2799, 2023.

[45] James Prather, Brent N. Reeves, Paul Denny, Brett A. Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. "it's weird that it knows what i want": Usability and interactions with copilot for novice programmers. *ACM Trans. Comput.-Hum. Interact.*, 31(1), nov 2023.

[46] replit. Meet ghostwriter, your partner in code., 2023.

[47] Steven I Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D Weisz. The programmer's assistant: Conversational interaction with a large language model for software development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces*, pages 491–514, 2023.

[48] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[49] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. An analysis of the automatic bug fixing performance of chatgpt. *arXiv preprint arXiv:2301.08653*, 2023.

[50] Claudio Spiess, David Gros, Kunal Suresh Pai, Michael Pradel, Md Rafiqul Islam Rabin, Susmit Jha, Prem Devanbu, and Toufique Ahmed. Quality and trust in llm-generated code. *arXiv preprint arXiv:2402.02047*, 2024.

[51] Priyan Vaithilingam, Tianyi Zhang, and Elena 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*, pages 1–7, 2022.

[52] Helena Vasconcelos, Gagan Bansal, Adam Fourney, Q Vera Liao, and Jennifer Wortman Vaughan. Generation probabilities are not enough: Exploring the effectiveness of uncertainty highlighting in ai-powered code completions. *arXiv preprint arXiv:2302.07248*, 2023.

[53] Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, et al. Recode: Robustness evaluation of code generation models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13818–13843, 2023.

[54] Tongshuang Wu, Kenneth Koedinger, et al. Is ai the better programming partner? human-human pair programming vs. human-ai pair programming. *arXiv preprint arXiv:2306.05153*, 2023.

[55] Weixiang Yan, Haitian Liu, Yunkun Wang, Yunzhe Li, Qian Chen, Wen Wang, Tingyu Lin, Weishan Zhao, Li Zhu, Shuiguang Deng, et al. Codescope: An execution-based multilingual multitask multidimensional benchmark for evaluating llms on code understanding and generation. *arXiv preprint arXiv:2311.08588*, 2023.

[56] John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *Advances in Neural Information Processing Systems*, 36, 2024.

[57] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. Large language models meet nl2code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7443–7464, 2023.

[58] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-bench and chatbot arena. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023.

[59] Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K. Reddy. Xlcost: A benchmark dataset for cross-lingual code intelligence, 2022.

[60] Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 21–29, 2022.

Checklist

1. For all authors...
    - (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] Contributions described at the end of Section 1 correspond to the sections that follow.
    - (b) Did you describe the limitations of your work? [Yes] See Section 6.
    - (c) Did you discuss any potential negative societal impacts of your work? [Yes] See Section 6.
    - (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
    - (a) Did you state the full set of assumptions of all theoretical results? [N/A]
    - (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments (e.g. for benchmarks)...
    - (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] We include code for the interface in supplementary material and instructions from the user study in Appendix B.
    - (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [N/A] We did not train any models.
    - (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] All figures in the paper have error bars.
    - (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [N/A] We only made API calls to externally hosted models.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
    - (a) If your work uses existing assets, did you cite the creators? [Yes] In Section 4, we cite all models used in the study.
    - (b) Did you mention the license of the assets? [Yes] See Appendix D.
    - (c) Did you include any new assets either in the supplemental material or as a URL? [Yes] We provide a URL to the dataset we are releasing.
    - (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [Yes] See Appendix B.
    - (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [Yes] See Appendix B.
5. If you used crowdsourcing or conducted research with human subjects...

(a) Did you include the full text of instructions given to participants and screenshots, if applicable? [Yes] See Section B.

(b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [Yes] See Appendix B for potential participant risks. We include copies of IRB approvals in the supplementary material.

(c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [Yes] See Section 4.

A Comparison to prior studies

Table 1: A comparison of our study against prior studies understanding programmer-LLM interactions in terms of the number of participants, models, types of LLM interaction, and tasks. Note that Cui et al. [14] was a field experiment and thus not a controlled user study with a fixed number of tasks.

Study	**# participants**	**# models**	**Autocomplete?**	**Chat?**	**# tasks**
Vaithilingam et al. [51]	24	1	✓	✗	3
Peng et al. [43]	95	1	✓	✗	1
Barke et al. [4]	20	1	✓	✗	4
Prather et al. [45]	19	1	✓	✗	1
Mozannar et al. [36]	21	1	✓	✗	8
Vasconcelos et al. [52]	30	1	✓	✗	3
Cui et al. [14]	1974	1	✓	✗	*
Ross et al. [47]	42	1	✗	✓	4
Chopra et al. [12]	14	1	✗	✓	4
Gu et al. [20]	22	1	✗	✓	10
Kazemitabaar et al. [25]	69	1	✗	✓	45
Nam et al. [39]	32	1	✗	✓	2
Ours	213	6	✓	✓	17

In Table 1, we compare the aspects of our study with prior works that have conducted user studies where programmers code with LLM support. To our knowledge, ours is the first study to consider models of varying performance capabilities and multiple forms of support. Additionally, we note that the majority of studies have similar participant profiles as ours (i.e., students with some programming experience and industry professions), though a few focus exclusively on novice programmers [25, 45]. Finally, multiple studies have limited scope in terms of the number and types of coding tasks that are considered (e.g., focusing on one minesweeper game [45] or simple plotting tasks [47]), which differ from the breadth of tasks that have been evaluated in benchmarks and are present in practical use cases.

B User study details

B.1 `RealHumanEval` interface screenshots

We show examples of the `RealHumanEval` web interface used in the study: autocomplete conditions (Figure 5 and Figure 6) and chat conditions (Figure 7). Note that the interface is the same as that of the autocomplete conditions for the no LLM condition, except there is no LLM to provide any inline code suggestions.

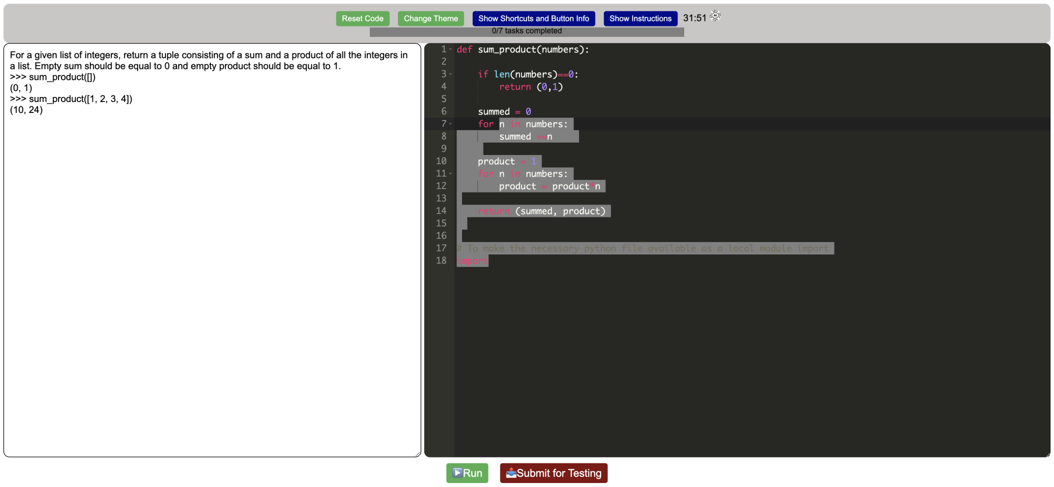
B.2 User Study Instructions

Before participants enter the main interface, they are provided with the following text:

> After you fill out the information here, click the Start Experiment button to proceed.
> Please DO NOT refresh or press back as you may lose a fraction of your progress, if needed you can refresh while coding but you will lose your code.
> Your name and email will NOT be shared with anyone or used in the study.
> Note that there is a chance the interface may not have AI, that is not a bug.
> By performing this task, you consent to share your study data.

In all conditions, a pop-up is displayed with the following instruction:

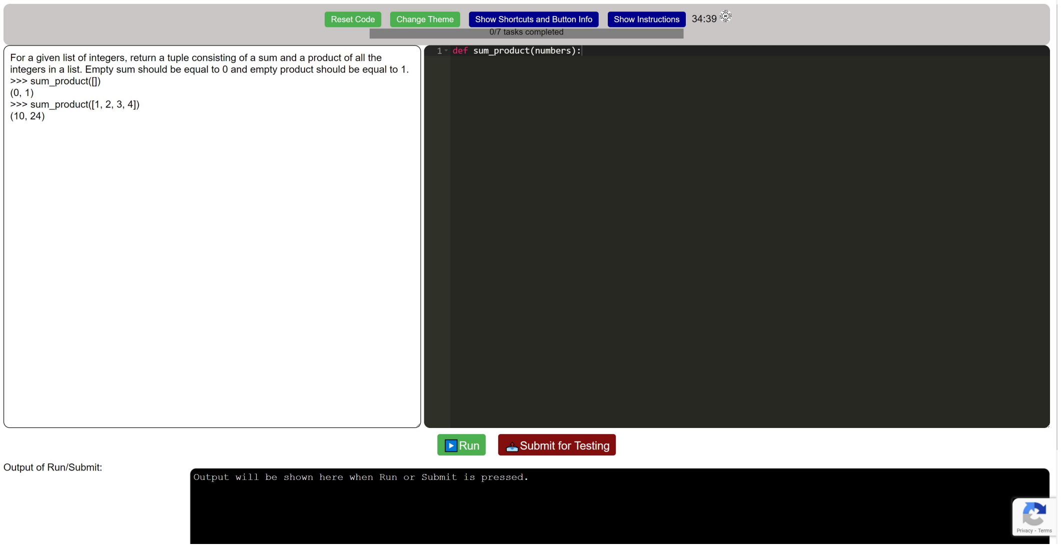
Welcome to the user study! You will first complete a tutorial task to make you familiar with the study.



For a given list of integers, return a tuple consisting of a sum and a product of all the integers in a list. Empty sum should be equal to 0 and empty product should be equal to 1.
>>> sum_product([])
(0, 1)
>>> sum_product([1, 2, 3, 4])
(10, 24)

```
def sum_product(numbers):

    if len(numbers)==0:
        return (0,1)

    summed = 0
    for n in numbers:
        summed +=n

    product = 1
    for n in numbers:
        product = product*n

    return (summed, product)


# To make the necessary python file available as a local module import
import
```

Figure 5: Screenshot of the autocomplete LLM-assistance interface in our user study.



For a given list of integers, return a tuple consisting of a sum and a product of all the integers in a list. Empty sum should be equal to 0 and empty product should be equal to 1.
>>> sum_product([])
(0, 1)
>>> sum_product([1, 2, 3, 4])
(10, 24)

```
def sum_product(numbers):
```

```
Output will be shown here when Run or Submit is pressed.
```

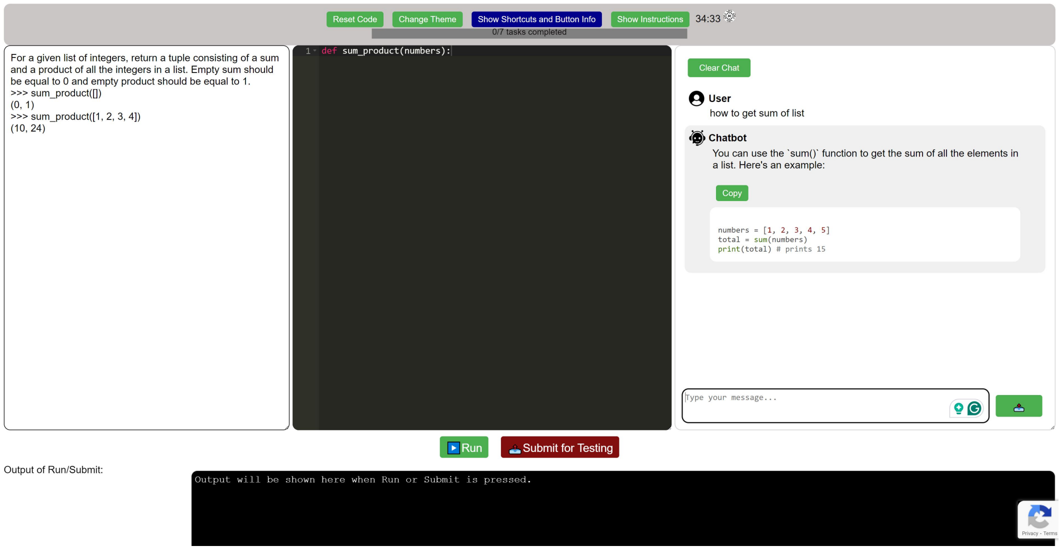
Figure 6: Another screenshot of the autocomplete LLM-assistance interface in our user study.

- You will be writing code in Python only and use only standard python libraries and only numpy and pandas.
- After the tutorial task, you will have 35 minutes total where you will try to solve as many coding tasks as possible one at a time.
- It is NOT allowed to use any outside resources to solve the coding questions (e.g., Google, StackOverflow, ChatGPT), your compensation is tied to effort only.

B.2.1 Autocomplete Condition

You will write code in the interface above: a code editor equipped with an AI assistant that provides suggestions inline.

- The AI automatically provides a suggestion whenever you stop typing for more than 2 seconds.
- You can accept a suggestion by pressing the key `[TAB]` or reject a suggestion by pressing `[ESC]`.



For a given list of integers, return a tuple consisting of a sum and a product of all the integers in a list. Empty sum should be equal to 0 and empty product should be equal to 1.

```
>>> sum_product([])
(0, 1)
>>> sum_product([1, 2, 3, 4])
(10, 24)
```

```
def sum_product(numbers):
```

User
how to get sum of list

Chatbot
You can use the `sum()` function to get the sum of all the elements in a list. Here's an example:

```
numbers = [1, 2, 3, 4, 5]
total = sum(numbers)
print(total) # prints 15
```

```
Output will be shown here when Run or Submit is pressed.
```

Figure 7: Screenshot of the chat LLM-assistance interface in our user study.

- You can also request a suggestion at any time by pressing `[CTRL+ENTER]` (Windows) or `[CMD+ENTER]` (Mac).
- You can run your code by pressing the run button and the output will be in the output box at the bottom in grey.
- **Press the submit button to evaluate your code for correctness. You can submit your code as many times as you wish until the code is correct.**
- If you cannot solve one of the tasks in 10 minutes, a button "Skip Task", only press this button if you absolutely cannot solve the task.

Note: please be aware that the AI assistant is not perfect and may provide incorrect suggestions. Moreover, the AI may generate potentially offensive suggestions especially if prompted with language that is offensive.

B.2.2 Chat Condition

You will write code in the interface above: a code editor equipped with an AI assistant chatbot in the right panel.

- The AI chatbot will respond to messages you send and incorporate previous messages in its response. The AI does not know what the task is or the code in the editor.
- When the AI generates code in its response, there is a COPY button that will show up above the code segment to allow you to copy.
- You can test your code by pressing the run button and the output will be in the output box at the bottom in grey.
- **Press the submit button to evaluate your code for correctness. You can submit your code as many times as you wish until the code is correct.**
- If you cannot solve one of the tasks in 10 minutes, a button "Skip Task", only press this button if you absolutely cannot solve the task.

Note: please be aware that the AI assistant is not perfect and may provide incorrect suggestions. Moreover, the AI may generate potentially offensive suggestions especially if prompted with language that is offensive.

B.2.3 No LLM Condition

You will write code in the interface above: a code editor.

- You can run your code by pressing the run button and the output will be in the output box at the bottom in grey.
- **Press the submit button to evaluate your code for correctness. You can submit your code as many times as you wish until the code is correct.**
- If you cannot solve one of the tasks in 10 minutes, a button "Skip Task", only press this button if you absolutely cannot solve the task.

B.2.4 Post-Study Questionnaire

- Thinking of your experience using AI tools outside of today's session, do you think that your session today reflects your typical usage of AI tools?
- How mentally demanding was the study? (1-20)
- How physically demanding was the study? (1-20)
- How hurried or rushed was the pace of the study? (1-20)
- How successful were you in accomplishing what you were asked to do? (1-20)
- How hard did you have to work to accomplish your level of performance? (1-20)
- How insecure, discouraged, irritated, stressed, and annoyed were you? (1-20)
- Overall, how useful/helpful was the AI assistant? (1-10)
- In which ways was the AI assistant helpful? What did it allow you to accomplish? (free-text)
- How could the AI suggestions be improved? (free-text)
- Additional comments (Optional): anything went wrong? any feedback? (free-text)

To ensure consistency in responses to scale-based questions, we labeled 1 with "low" and either 10 or 20 with "high" depending on the question.

B.3 Data release considerations

We took the following measures to mitigate potential ethical concerns regarding the release of the study. First, the study protocol was approved by institutional IRB review. Second, before participating in the actual study, all participants were provided with a consent form outlining the study and the data that would be collected as part of the study (including interaction data with LLMs) and provided with the option to opt out of the study if they so choose. Finally, after data collection and prior to public data release, the authors carefully checked all participant interactions with LLMs, particularly chat dialogue, to ensure that no personally identifiable information was revealed.

C Task Design

C.1 Task categories

Algorithmic coding problems: Many coding tasks require programmers to implement algorithmic thinking and reasoning and are widely used to evaluate programmers in coding interviews. To identify algorithmic coding problems, we sample representative problems from the HumanEval dataset [10]. Given `gpt-3.5-turbo`'s high performance on this type of problem, we ensure that we also include problems where it fails to solve the problem on its own. We evaluated each question using test cases from the HumanEval dataset. We included the following problem ids from HumanEval: is_bored 91, is_multiply_prime 75, encode_message 93, count_nums 108, order_by_points 145, even_odd_count 155, sum_product 8, triple_sum_to_zero 40. In addition, we created a custom problem called event_scheduler. All tasks with unit tests will be released.

Editing and augmenting existing code: When working with existing repositories, programmers will often need to edit and build on code that may have been written by others [49]. We designed questions where participants are either provided with either code scaffold to fill in or with code body that they are asked to modify the functionality of. When designing such questions, we take care to avoid common implementations (e.g., a traditional stack and queue) that would have appeared in LLM training data. We also constructed a set of test cases for each question. The four problem names are calculator, tokenizer, login authenticator and retriever.

For example, here is the login authenticator problem description:

> Your goal is to implement the `LoginAuthenticator` class, which will be used to authenticate users of a system. The class will include the following methods:
>
> _hash_password (Private Method): Creates a hash of a given password. Accepts a *password* (string) and returns the hashed password using any hashing technique.
>
> add_user Method: Adds a new user to the system with a username and a password. It checks if the username already exists, hashes the password if it does not, and stores the credentials. Returns True if successful.
>
> remove_user Method: Removes a user from the system by deleting their username entry from `self.user_credentials` if it exists. Returns True if successful.
>
> change_password Method: Changes a user's password after authenticating the user with their old password. If authenticated, it hashes the new password and updates `self.user_credentials`. Returns True if successful.

The programmer is given the following initial code:

```
class LoginAuthenticator:
    def __init__(self):
        # DO NOT CHANGE
        self.user_credentials = {}  # dictionary for username:
                                          hashed_password

    def _hash_password(self, password):
        # WRITE CODE HERE
        return

    def add_user(self, username, password):
        # WRITE CODE HERE
        return

    def authenticate_user(self, username, password):
        # DO NOT CHANGE
        #Checks if the given username and password are valid
        if username not in self.user_credentials:
            return False
        return self.user_credentials[username] == self._hash_password(
                                          password)

    def remove_user(self, username):
        # WRITE CODE HERE
        return

    def change_password(self, username, old_password, new_password):
        # WRITE CODE HERE
        return
```

Data science tasks: Given the increased usage of data in many engineering disciplines, programmers are often involved in data science problems. We design data science problems inspired by the

DS-1000 dataset [28], where participants need to perform *multiple* data manipulation and wrangling operations and return a resulting Pandas dataframe. To ensure that an LLM cannot achieve perfect accuracy on its own, we only show an example of the input and target dataframes without providing specific instructions on each operation. The code will be evaluated based on the correctness of the dataframe in an element-wise fashion. The four problem names are table_transform_named, table_transform_unnamed1, table_transform_unnamed2 and t_test.

Here is for example the problem table_transform_unnamed1:

> Given the input pandas DataFrame:
>
> | | col1 | col2 | col3 | col4 | col5 |
> |---|---|---|---|---|---|
> | 0 | 6 | 1 | 5.38817 | 3 | 2 |
> | 1 | 9 | 2 | 4.19195 | 5 | 8 |
> | 2 | 10 | 8 | 6.8522 | 8 | 1 |
> | 3 | 6 | 7 | 2.04452 | 8 | 7 |
> | 4 | 1 | 10 | 8.78117 | 10 | 10 |
>
> Transform this DataFrame to match the following output structure, recognizing the relationship between the input and output DataFrames:
>
> | | col1 | col2 | col3 |
> |---|---|---|---|
> | 0 | 6 | 2 | 8.38817 |
> | 1 | 15 | 3 | 9.19195 |
> | 2 | 25 | 9 | 14.8522 |
> | 3 | 31 | 8 | 10.0445 |
> | 4 | 32 | 11 | 18.7812 |
> | 0 | 0 | 0 | 0 |
> | 0 | 0 | 0 | 0 |
>
> Implement a function named `transform_df` that takes the input DataFrame and returns the transformed DataFrame, discovering and applying the patterns between them.

The programmer is given the following initial code:

```python

import pandas as pd
from io import StringIO

# Original dataset
data = '''
col1,col2,col3,col4,col5
6,1,5.3881673400335695,3,2
9,2,4.191945144032948,5,8
10,8,6.852195003967595,8,1
6,7,2.0445224973151745,8,7
1,10,8.781174363909454,10,10
'''

# Read the dataset into a DataFrame
df = pd.read_csv(StringIO(data))

def transform_df(df):
    # Your code here

print(transform_df(df))
```

C.2 Task organization

We created five task sets where we fixed the first task (in addition to the tutorial sum_product task) and varied the remaining tasks randomly ensuring a split across the categories. The five sets are:

1. Task Set 1: even_odd_count, triple_sum_to_zero, table_transform_named, tokenizer, encode_message, t_test, event_scheduler.
2. Task Set 2: even_odd_count, is_bored, login_authenticator, is_multiply_prime, count_nums, table_transform_named, calculator.
3. Task Set 3: even_odd_count, count_nums, calculator, table_transform_unnamed2, login_authenticator, encode_message, is_bored.
4. Task Set 4: even_odd_count, order_by_points, retriever, triple_sum_to_zero, tokenizer, event_scheduler, encode_message.
5. Task Set 5: even_odd_count, is_multiply_prime, table_transform_unnamed1, t_test, is_bored, order_by_points, triple_sum_to_zero.

D LLM Details

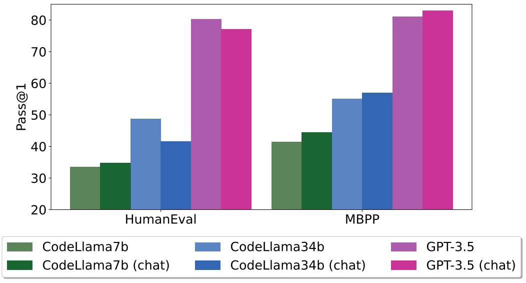


Figure 8: `Pass@1` of LLM models and their chat variants on two canonical benchmarks, HumanEval and MBPP (results from [48, 33]), showing that `CodeLlama-7b` models perform worse than `CodeLlama-34b` models, which are less performant than `GPT-3.5` models.

We select three models of varying benchmark performance as shown in Figure 8. Here we provide links to model weights (where applicable) and any additional details.

- **CodeLlama (7b, 34b) and CodeLlama Instruct (7b, 34b).** Accessed from `https://api.together.xyz/`. Note that the base model variants are no longer available from this source. The license for the CodeLlama models is at `https://github.com/meta-llama/llama/blob/main/LICENSE`.
- **GPT-3.5-turbo.** Specific model version is `gpt-3.5-turbo-0613`. Accessed through the OpenAI API. This is a closed model and does not have an associated license.
- **GPT-3.5-turbo-instruct.** Accessed through the OpenAI API. This is a closed model and does not have an associated license.

LLM parameters. For all LLMs, we used a temperature setting of 1 to ensure varied responses. For autocomplete LLMs, each time we query the LLM to generate a suggestion, we sample a random number according to a normal distribution with mean 64 tokens and std 15 truncated to the range [10,120] and set the max_token parameter to that sampled value. We used the mean value of 64 in accordance with Personal Copilot HuggingFace implementation [4]. We allow the max_token length to be random so that we have access to future data to determine the optimal length of suggestions, this is because base LLMs are not trained with an EOS token and thus do not know when to stop generating code. For the chat LLMs, we set the max_token parameter to 512 tokens constant.

---

[4] `https://huggingface.co/blog/personal-copilot`

Why we did not select other model candidates. Of the CodeLlama models available to use at the time of the study, we omitted CodeLlama-13b. We did not select CodeLlama-13b as its performance on HumanEval is very similar to the 7b variant. Additionally, CodeLlama-70b had not been released when we conducted the study. We did not include GPT-4 because of the lack of availability of the completion-based variant via API.

D.1 Prompts used

We used the following system prompt for all chat-based LLMs:

```
You are an expert Python programmer, be helpful to the user
and return code only in Python.
```

For autocomplete-based LLMs, the first line of the prompt is always the following:

```
# file is main.py, ONLY CODE IN PYTHON IN THIS FILE
```

These prompts help to ensure that LLM responds in Python.

E Additional Results

E.1 Pre-registration

We pre-registered our study design prior to data collection but not the analysis plan `https://aspredicted.org/blind.php?x=K3P_K1J`. Due to the limit on the number of participants who completed the task within the timeframe of the study, we thus ended up with fewer participants in the final dataset than we originally anticipated being able to collect (i.e., 30 per condition instead of 50 per condition). As a result, we opted to pool together data from the same model class to study both hypotheses. All other additional analyses in this work are exploratory and were not pre-registered.

E.2 Dataset Analysis

We post-processed both datasets to ensure they did not reveal any identifying information about participants or contain harmful language.

Autocomplete dataset. Recall that users had the option to request suggestions via hotkey or were provided the suggestion after some time. As shown in Figure 9, participants are much more likely to accept suggestions if they request them. Interestingly, `CodeLlama-34b` suggestions seemed to be more preferred than `CodeLlama-7b` when requested.

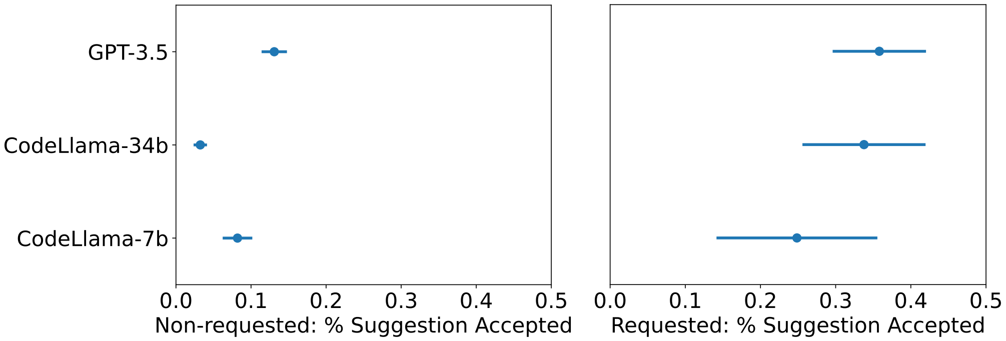


Figure 9: Comparing the acceptance rate for when participants requested suggestions with when they were automatically provided with suggestions by the autocomplete system.

Chat dataset. We analyze the 775 chat messages participants sent across the three conditions, as shown in Figure 10. On average 2.8 messages were sent per task with a length of 100.8 characters. We note that there is a particularly long tail in terms of words appearing in chat messages because many questions contained implementation-specific variables. In accordance with our findings that LLMs were most useful for data manipulation tasks, we also find that participants engaged with LLM support the most for those tasks.

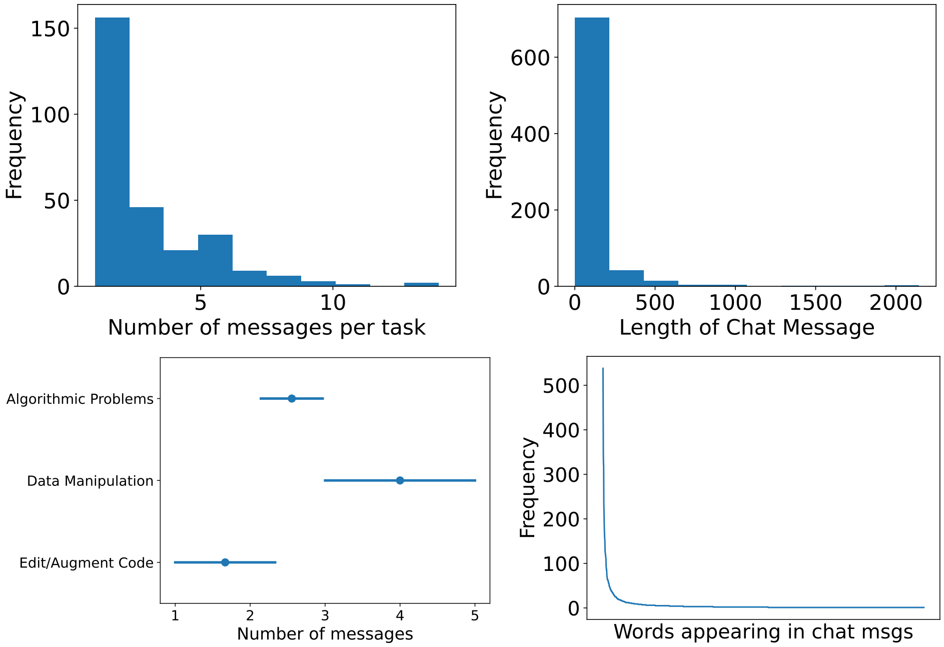


Figure 10: Analysis of the number of messages sent per task (top left), the length of chat messages (top right), the number of messages sent per task category (lower left), and the frequency of words appearing in chat messages (lower right).

E.3 Accounting for task difficulty

To facilitate comparisons between different sets of tasks, which may have varying difficulty, the value of each metric is z-scored within the task set:

$$M^z_{i,t} = \frac{M_{i,t} - \mu_{M,t}}{\sigma_{M,t}}$$

where $M^z_{i,t}$ is the value of metric M achieved by participant i, z-scored within task set t; $\mu_{M,t}$ and $\sigma_{M,t}$ are the mean and standard deviation of metric M for task set t, across all participants. We rerun our analysis for performance metrics and present results in Figure 11.

E.4 Task completion time

In Figure 3, we find the most significant differences between models in terms of task completion time. We further analyze task completion time across multiple axes.

By task type. When comparing when participants have access to LLM assistance versus the control condition, as shown in Figure 12, we find suggestive evidence that LLM assistance was particularly effective in reducing the time programmers needed to solve data manipulation tasks and problems that required editing and augmenting existing code, but not for algorithmic problems. We also analyze whether participants benefited from LLM assistance on an individual task level in Figure 13. We

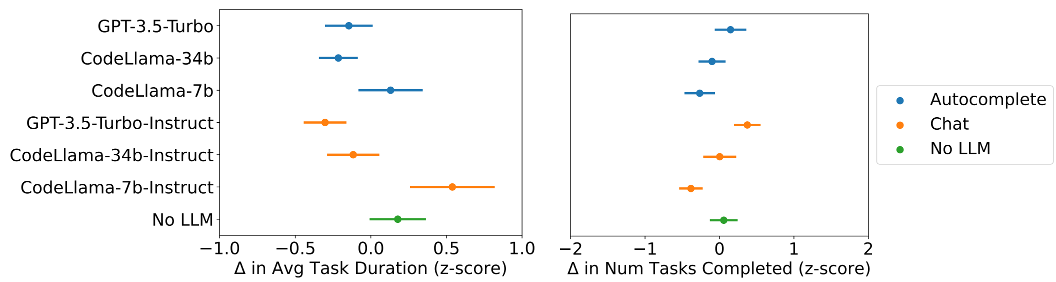


Figure 11: Performance results across models, z-scored to account for potential variation in task difficulty across sets.

observe that trends for individual tasks within a category are similar, indicating the importance of understanding how programmers interact with LLMs for different *types* of tasks.

Verifying outlier behavior. We plot a histogram of task completion times in Figure **??** to verify that across participants, there were not a significant number of outliers. We also performed a similar check by plotting across conditions in Figure 16 to ensure that there was not differing behavior across participants (e.g., no bimodal behavior within a given condition).

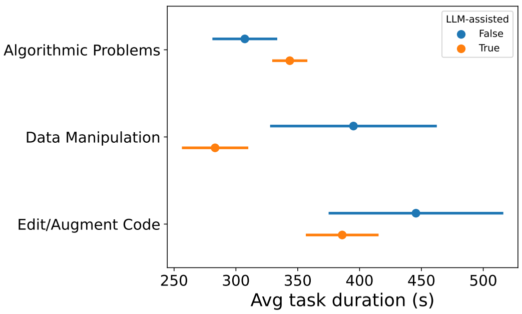


Figure 12: Average task duration with and without LLM assistance with standard error by task category.

E.5 Code Quality Metrics

Code Comments. Code written with the assistance of the LLM will inherit some of the characteristics of the writing style of the LLM. One instance of that is comments in the code written. We investigate the number of comments written by participants for the different types of interaction with the LLM: autocomplete, chat, or no LLM. We count how many additional comments are in the code participants write compared to the number of comments in the provided code participants complete. Participants in the autocomplete conditions wrote 0.85 ± 0.1 additional comments, in the chat condition wrote 0.59 ± 0.08 comments and those in the `No LLM` condition wrote 0.41 ± 0.13 comments. Participants writing code with autocomplete LLM write twice as many comments as those without an LLM ($p = 3e - 6$). There are two possible explanations for this increase: first, programmers usually prompt the LLM with inline comments to get a suggestion they desire, and second, we often observe that code generated by LLMs is often heavily commented. This indicates that we can potentially differentiate code written by programmers with LLM assistance by the number of comments in the code.

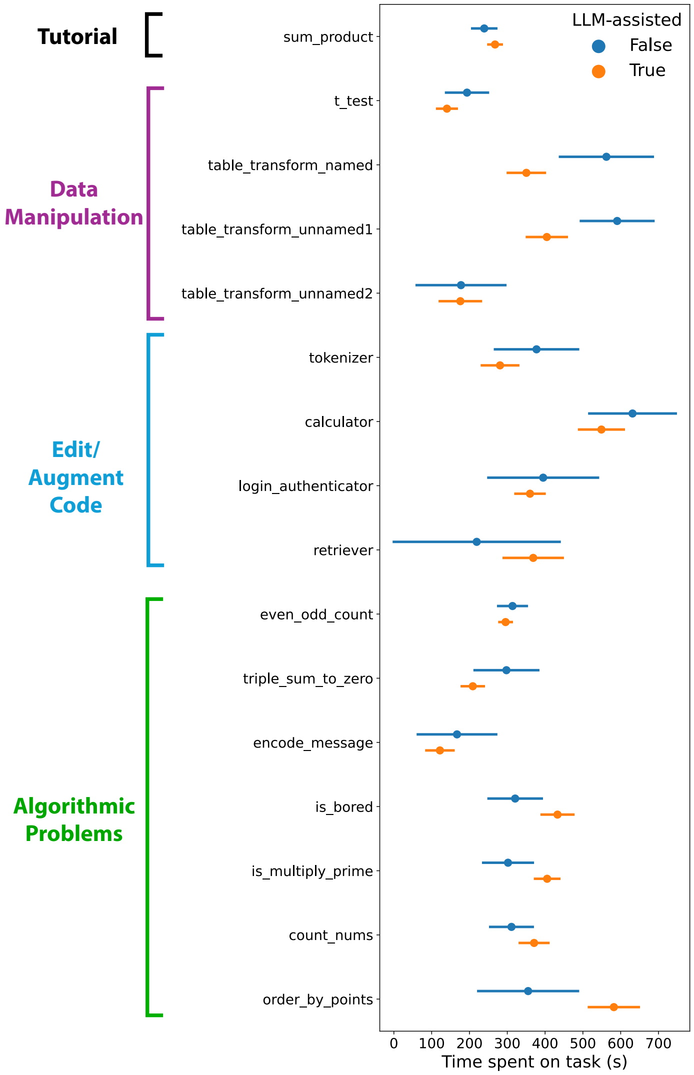


Figure 13: Time to task completion with and without LLM assistance, reported by task and grouped by task category, with standard error.

F Design Opportunities

To understand the design opportunities around improving the coding assistance provided through `RealHumanEval`, we analyzed a post-study question on how coding assistants could be improved. Answers to the question were collected in free response format and were optional, though it was answered by the majority of participants (174 out of the 213). We summarize participant suggestions into general comments that could apply to both types of interactions and identify autocomplete- and chat-specific suggestions.[5]

Both autocomplete and chat models need improved context. A theme that spanned both types of interactions and model types was the perceived lack of context that the LLM had about the general task when providing either suggestions or chat responses (example shown in Figure 17). While one might expect that a more performant model might mitigate these concerns, we do not observe a significant decrease in mentions regarding this issue for `GPT-3.5` models compared to both `CodeLlama-7b`

---

[5] We omit the obvious, blanket suggestion for replacing the assistant with a better LLM, as model-only performance is one of the independent variables in our experiment and a more performant model would undoubtedly improve the assistance provided.

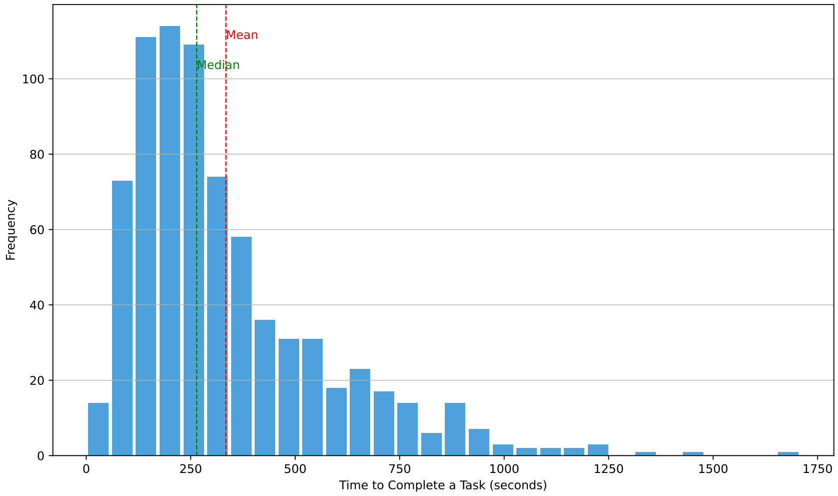


Figure 14: Histogram depicting the distribution of task completion times across all participants and conditions. The histogram is overlaid with dashed lines representing key statistical measures: the mean (red) and the median (green).

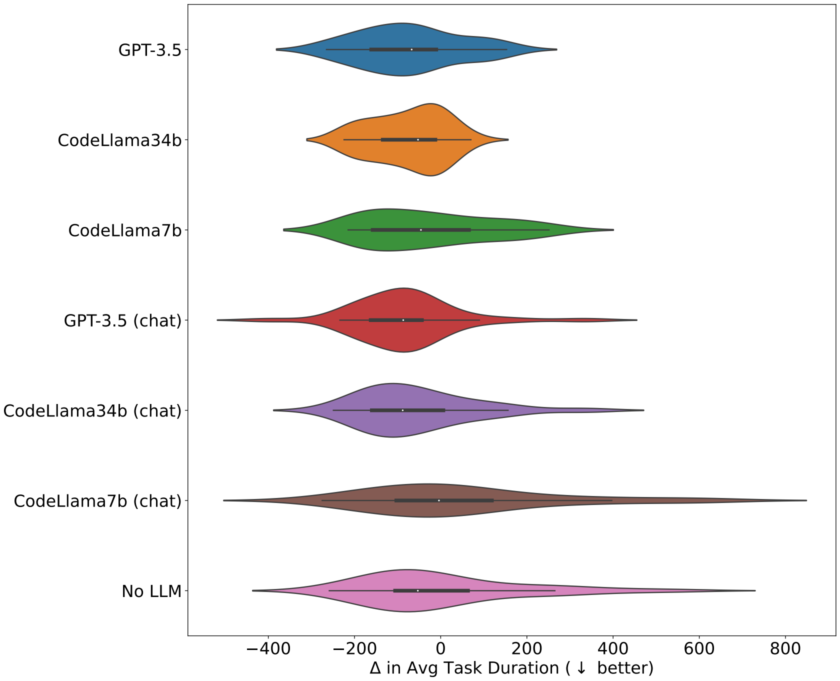


Figure 15: Violin plot of the difference in average task duration times (in seconds) between the No-LLM condition and all other conditions.

and `CodeLlama-34b` models. In general, it may not be obvious how to concisely specify the full "context"—recall that we intentionally considered a set-up where the LLM is unaware of task T to mimic realistic constraints—but the development of new interfaces to facilitate context specification and mechanisms to prompt for additional task-specific information could improve LLM generations. Additionally, further baseline checks can be implemented to minimize concerns mentioned by participants (e.g., ensuring that the LLM responses are provided in the correct programming language, beyond prompting-based approaches implemented in our study). We note that issues surrounding context control have also been highlighted in prior work [12, 4].

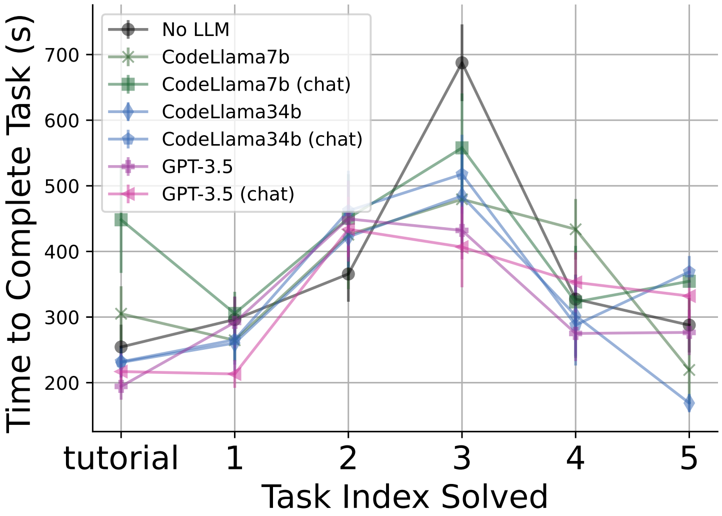
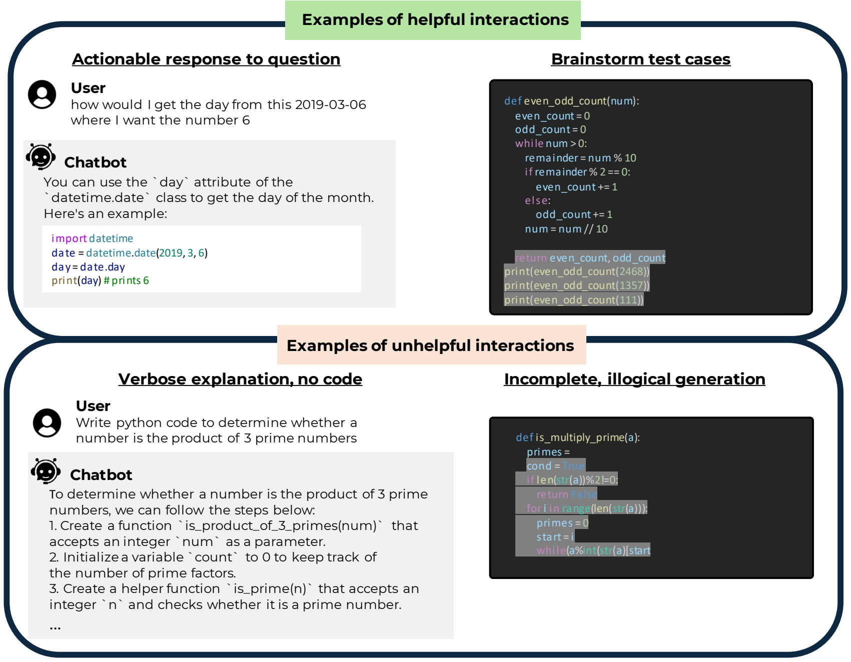


Figure 16: For each of the seven conditions, we plot the average time for participants to complete the tutorial task, the first task they solved, the second task they solved, and so on.

Autocomplete-specific suggestions. We highlight the three most commonly mentioned avenues of improvement across all three model types. *(1) Minimize suggestion frequency:* Participants noted that the frequency of suggestions appearing in the code editor could disrupt their train of thought. To address this issue, it may be preferable to allow participants to turn off the LLM model when they are brainstorming the next steps or to modify the LLM to detect when participants may not need as frequent suggestions based on their current coding behavior. Moreover, we observe quantitatively that participants are between $3 - 10\times$ more likely to accept an assistant's suggestion *if* they requested it, as shown in Figure 9. *(2) Dynamic suggestion length:* A common issue with autocomplete interactions noted by participants was the presence of "incomplete variable definitions or function implementations" and "fragmented code" (e.g., Figure 18 (left)). As this behavior is a product of the fixed length of LLM generations, autocomplete assistants can be improved by ensuring the suggestion is complete before terminating generation. *(3) More concise suggestions:* Finally, participants also noted that code completions could be more concise, as "it was overwhelming" and "large chunks of code... start deviating from the task question" (e.g., Figure 18 (right)). It is an open question to determine the appropriate length for how much code to generate in a context-aware manner.

Chat-specific suggestions. There were three common suggestions shared across models. *(1) Responses should focus on code, rather than explanation:* It is well known that chat LLMs tend to generate verbose responses, which could be detrimental when used as programming assistants. An example of a lengthy response is in Figure 20. In particular, participants noted the additional time required to read large blocks of texts and suggested to "get rid of all explanations and stick to code only, unless the user specifies they want explanations." Additionally, when focusing on code, participants suggested that the chat assistant could anticipate alternative implementations *(2) Improved dialogue experience:* First, instead of making assumptions about potentially ambiguous points in a programmer's question (e.g., as in Figure 19), a participant suggested that the LLM "could ask clarifying questions or provide multiple suggestions." Additionally, in particular for `CodeLlama-7b`, participants asked for better consistency across multiple chat messages (e.g., "It wasn't able to refer back to previous messages that I had sent when answering a question."). *(3) Better integration with code editor:* Currently, the burden is on the programmer to appropriately prompt the chat assistant with questions and then to integrate chat suggestions into the code body in the editor. This onus can be reduced by more readily incorporating "the code and the most recent error, if any, as well as the test case that generated it in the context for the assistant" and "autocorrect code" based on its suggestions.



Examples of helpful interactions
Actionable response to question
Brainstorm test cases

User
how would I get the day from this 2019-03-06 where I want the number 6

Chatbot
You can use the `day` attribute of the `datetime.date` class to get the day of the month. Here's an example:

```
import datetime
date = datetime.date(2019, 3, 6)
day = date.day
print(day) # prints 6
```

```
def even_odd_count(num):
  even_count = 0
  odd_count = 0
  while num > 0:
    remainder = num % 10
    if remainder % 2 == 0:
      even_count += 1
    else:
      odd_count += 1
    num = num // 10

  return even_count, odd_count
print(even_odd_count(2468))
print(even_odd_count(1357))
print(even_odd_count(111))
```

Examples of unhelpful interactions
Verbose explanation, no code
Incomplete, illogical generation

User
Write python code to determine whether a number is the product of 3 prime numbers

Chatbot
To determine whether a number is the product of 3 prime numbers, we can follow the steps below:
1. Create a function `is_product_of_3_primes(num)` that accepts an integer `num` as a parameter.
2. Initialize a variable `count` to 0 to keep track of the number of prime factors.
3. Create a helper function `is_prime(n)` that accepts an integer `n` and checks whether it is a prime number.
...

```
def is_multiply_prime(a):
  primes =
  cond = True
  if len(str(a))%2!=0:
    return False
  for i in range(len(str(a))):
    primes = 0
    start = i
    while(a%int(str(a)[start
```

Figure 17: Examples from of helpful and unhelpful chat and autocomplete interactions from the user study. While these examples showcase how LLM assistance can improve programmer productivity (e.g., by providing actionable responses and generating test cases), they also highlight how programmer-LLM interactions can be improved. We discuss design opportunities collected from post-task participant responses in Section F and provide more examples in Appendix G.

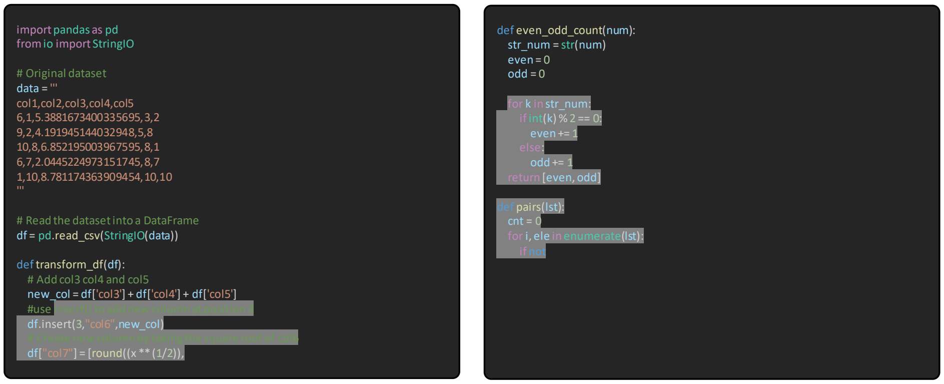
Why was `CodeLlama-34b` less preferred by users? Based on participants' survey responses, we identify two potential reasons that might qualitatively explain why `CodeLlama-34b` was less preferred for both autocomplete and chat. For autocomplete, the lack of context was a particularly prevalent issue in responses for `CodeLlama-34b`, mentioned by 54% of responses, as compared to 32% and 28% of `CodeLlama-7b` and `GPT-3.5` responses respectively. In particular, participants noted that the generated suggestions were often irrelevant to the prior code and in the wrong programming language. We show examples of rejected suggestions that illustrate a lack of context from participants who interacted with the `CodeLlama-34b` model in Figure 21. For chat, while there were no exceptional concerns about lack of context, `CodeLlama-34b` had the most mentions of latency as a point of improvement (6 mentions as compared to only 2 and 1 mentions for `CodeLlama-7b` and `GPT-3.5` respectively). For example, one participant noted that "the responses are slow so sometimes it was faster to go off of my memory even if I wasn't sure if it would work." Indeed, we found that `CodeLlama-34b` response time (about 10 seconds) was on average twice as slow as either `CodeLlama-7b` or `GPT-3.5` (about 5 seconds). We note that this slight delay did not significantly impact any participant's performance metrics.

F.1 Opportunities to use data

Simulating programmer-LLM interaction. The data collected in our study presents an opportunity to build and evaluate simulation environments that mimic how programmers write code with an LLM. Essentially, the simulator could be used to more efficiently replicate the results of `RealHumanEval` and evaluate a wider set of models. However, despite initial work on simulating programmer-LLM interaction [37], building a useful simulator requires significant training and validation. Our dataset provides training data for both chat and autocomplete interactions: The dataset of interactions with the chat models $\mathcal{D}_{\text{chat}}$ allows us to simulate the queries programmers make to the chat assistant given the code they have currently written. The dataset of interactions with the autocomplete models $\mathcal{D}_{\text{ac}}$ can allow us to simulate finer-grain interactions with LLM suggestions such as verifying and editing suggestions, among other activities outlined in [37]. To validate a proposed simulator, one should test whether it faithfully replicates the trends observed in `RealHumanEval` before it can be used as an evaluation benchmark.

Optimizing suggestions from human feedback. In addition to using the human feedback data to simulate the interaction, one can use it to fine-tune the models. For instance, the dataset of interactions with autocomplete models $\mathcal{D}_{\text{ac}}$ reveals which suggestions programmers accept and which they reject, which can be used to update the LLM and generate suggestions that maximize the probability of being accepted by the programmer. Moreover, the dataset also captures how accepted suggestions were edited over time, which can be used to generate suggestions that are more likely to persist in the programmer's code. Finally, an LLM that is not instruction-tuned usually requires specifying a maximum generation length parameter to stop the generation of a code suggestion. In our autocomplete implementation, we intentionally randomized the maximum suggestion length of the generated suggestion to be between the range $[10, 120]$ with a mean token length of 64. This design decision can provide yet another signal about when the LLM should stop generating code.

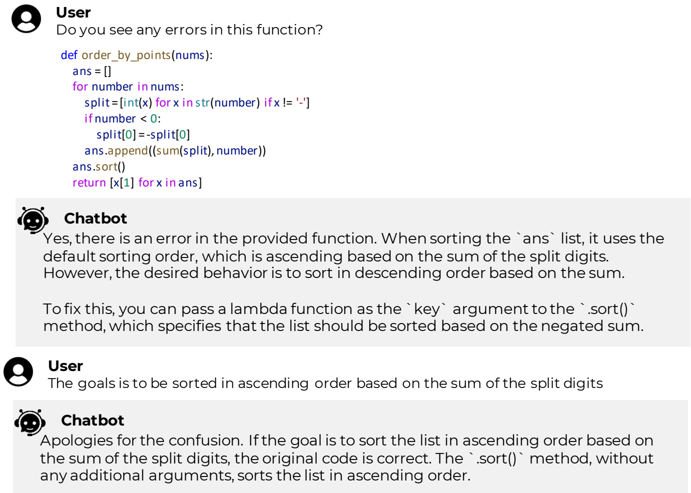
G Example user interactions



```
import pandas as pd
from io import StringIO

# Original dataset
data = '''
col1,col2,col3,col4,col5
6,1,5.3881673400335695,3,2
9,2,4.191945144032948,5,8
10,8,6.852195003967595,8,1
6,7,2.0445224973151745,8,7
1,10,8.781174363909454,10,10
'''

# Read the dataset into a DataFrame
df = pd.read_csv(StringIO(data))

def transform_df(df):
    # Add col3 col4 and col5
    new_col = df['col3'] + df['col4'] + df['col5']
    #use insert() to add new column at position 3
    df.insert(3,"col6",new_col)
    # Create new column by taking the square root of col6
    df["col7"] = [round((x ** (1/2)),
```

```
def even_odd_count(num):
    str_num = str(num)
    even = 0
    odd = 0

    for k in str_num:
        if int(k) % 2 == 0:
            even += 1
        else:
            odd += 1
    return [even, odd]

def pairs(lst):
    cnt = 0
    for i, ele in enumerate(lst):
        if not
```

Figure 18: Examples of problematic autocomplete suggestions: incomplete suggestion (left) and starting new irrelevant function (right).



User
Do you see any errors in this function?

```
def order_by_points(nums):
    ans = []
    for number in nums:
        split = [int(x) for x in str(number) if x != '-']
        if number < 0:
            split[0] = -split[0]
        ans.append((sum(split), number))
    ans.sort()
    return [x[1] for x in ans]
```

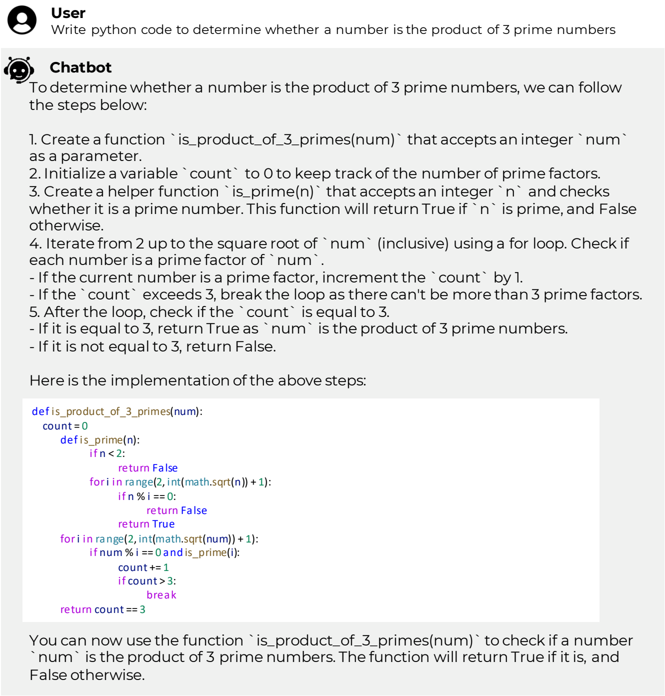
Chatbot
Yes, there is an error in the provided function. When sorting the `ans` list, it uses the default sorting order, which is ascending based on the sum of the split digits. However, the desired behavior is to sort in descending order based on the sum.

To fix this, you can pass a lambda function as the `key` argument to the `.sort()` method, which specifies that the list should be sorted based on the negated sum.

User
The goals is to be sorted in ascending order based on the sum of the split digits

Chatbot
Apologies for the confusion. If the goal is to sort the list in ascending order based on the sum of the split digits, the original code is correct. The `.sort()` method, without any additional arguments, sorts the list in ascending order.

Figure 19: Example of a chat interaction where the chat assistant could have proactively asked more clarifying questions. Note that the chat agent response is shortened (by excluding code snippet) for brevity.



User
Write python code to determine whether a number is the product of 3 prime numbers

Chatbot
To determine whether a number is the product of 3 prime numbers, we can follow the steps below:

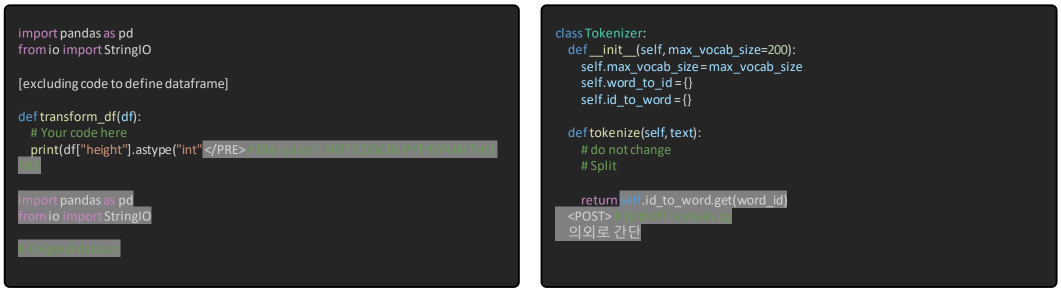
1. Create a function `is_product_of_3_primes(num)` that accepts an integer `num` as a parameter.
2. Initialize a variable `count` to 0 to keep track of the number of prime factors.
3. Create a helper function `is_prime(n)` that accepts an integer `n` and checks whether it is a prime number. This function will return True if `n` is prime, and False otherwise.
4. Iterate from 2 up to the square root of `num` (inclusive) using a for loop. Check if each number is a prime factor of `num`.
- If the current number is a prime factor, increment the `count` by 1.
- If the `count` exceeds 3, break the loop as there can't be more than 3 prime factors.
5. After the loop, check if the `count` is equal to 3.
- If it is equal to 3, return True as `num` is the product of 3 prime numbers.
- If it is not equal to 3, return False.

Here is the implementation of the above steps:

```
def is_product_of_3_primes(num):
  count = 0
      def is_prime(n):
            if n < 2:
                  return False
            for i in range(2, int(math.sqrt(n)) + 1):
                  if n % i == 0:
                        return False
                  return True
      for i in range(2, int(math.sqrt(num)) + 1):
            if num % i == 0 and is_prime(i):
                  count += 1
                  if count > 3:
                        break
      return count == 3
```

You can now use the function `is_product_of_3_primes(num)` to check if a number `num` is the product of 3 prime numbers. The function will return True if it is, and False otherwise.

Figure 20: Example of an overly lengthy response from a chat interaction.



```
import pandas as pd
from io import StringIO

[excluding code to define dataframe]

def transform_df(df):
   # Your code here
   print(df["height"].astype("int" </PRE> # file is a cell, NOT CODE IN PYTHON IN THIS
FILE

import pandas as pd
from io import StringIO

# Original dataset
```

```
class Tokenizer:
   def __init__(self, max_vocab_size=200):
      self.max_vocab_size = max_vocab_size
      self.word_to_id = {}
      self.id_to_word = {}

   def tokenize(self, text):
      # do not change
      # Split

      return self.id_to_word.get(word_id)
   <POST> # End of file main.py
   의외로 간단
```

Figure 21: Examples of rejected suggestions from `CodeLlama-34b`, which failed to consider the context of existing code: (left) the suggested code tried to import the same packages that are already present and (right) the suggested code trails off into irrelevant, non-Python text.