Simulating Iterative Human-AI Interaction in Programming with LLMs

Hussein Mozannar^{1,2} * **Valerie Chen**^{4*} **Dennis Wei**^{1,3} **Prasanna Sattigeri**^{1,3}
Manish Nagireddy^{1,3} **Subhro Das**^{1,3} **Ameet Talwalkar**^{4} **David Sontag**^{1,2}
^{1}MIT-IBM Watson AI Lab, Cambridge, MA
^{2}Massachusetts Institute of Technology, Cambridge, MA
^{3}IBM Research, Cambridge, MA
^{4} Carnegie Mellon University, PA
(Work In Progress)

Abstract

Large language models (LLMs) are increasingly used to support humans in tasks involving writing natural language and programming. How do we evaluate the benefits of LLM assistance for humans and learn from human interaction? We argue that benchmarks that evaluate the abilities of the model in isolation are not sufficient to reveal its impact on humans. Ideally, we can conduct user studies where humans complete tasks with the LLM and measure outcomes of interest. However, this can be prohibitively expensive in terms of human resources, especially as we want to iterate on model design continuously. We propose building a simulation environment that mimics how humans interact with the LLM, focusing in this work on assistants that provide inline suggestions for coding tasks. The environment simulates the multi-turn interactions that occur in programming with LLMs and uses a secondary LLM to simulate the human. We design the environment based on work that studies programmer behavior when coding with LLMs to make sure it is realistic. The environment allows us to evaluate the abilities of different scales of LLMs in terms of simulation metrics of success. The simulation also allows us to collect data that can be potentially used to improve the LLM's ability to assist humans, which we showcase with a simple experiment.

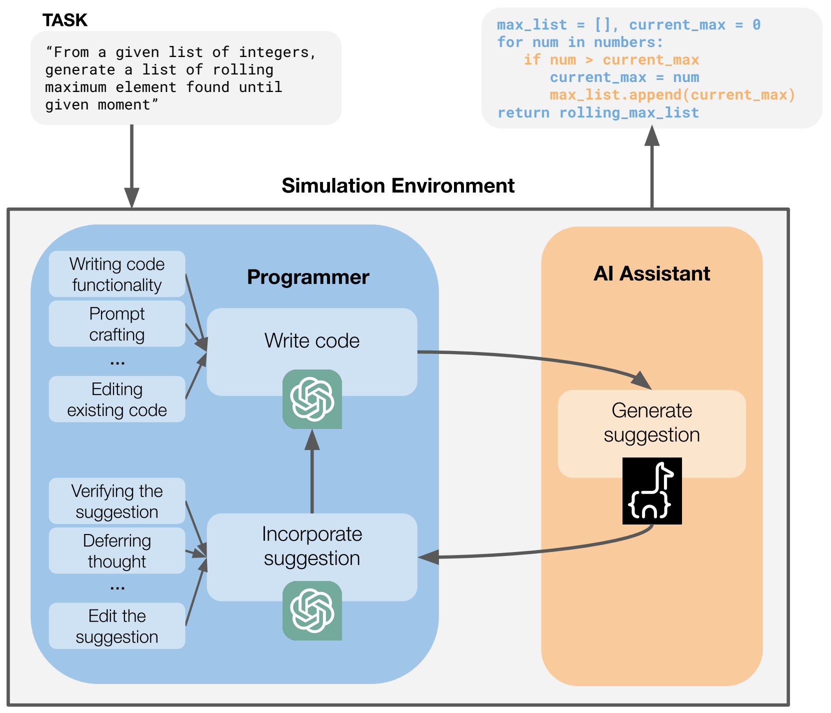
1 Introduction

The emergence of AI assistants powered by large language models (LLMs), such as ChatGPT [16] and Copilot [8], holds the potential to enhance productivity in various industries. In contrast with traditional prediction models, LLMs are thought to be well-suited to support humans in generative, language-based tasks, which include writing [12] and coding [13]. In the context of generative tasks, human interactions with AI assistants typically involve a repeated set of interactions, where each round of interaction involves the human generating some amount of text, the AI providing some amount of suggested text, and then the human deciding whether to accept the suggested text and potentially making further edits. In this work, we focus on a canonical setting for coding tasks where programmers use AI assistants that provide inline suggestions. Examples of such AI assistants include GitHub's Copilot [8], Replit's Ghostwriter [20], and Amazon CodeWhisperer [1].

To evaluate the extent an AI assistant can support programmers, one could consider the AI's performance on coding benchmarks (e.g., HumanEval [5][2] or MBPP [2]), where the AI assistant is

*Correspondence to `mozannar@mit.edu` and `valeriechen@cmu.edu`

[2]Despite what its name may suggest, the HumanEval benchmark does not involve human evaluation.


TASK

"From a given list of integers, generate a list of rolling maximum element found until given moment"

```
max_list = [], current_max = 0
for num in numbers:
   if num > current_max
      current_max = num
      max_list.append(current_max)
return rolling_max_list
```

Figure 1: We propose an environment that takes as input a coding task and simulates how a programmer would iteratively interact with an AI assistant to accomplish the task. This simulation environment allows us to evaluate the ability of different models to theoretically assist humans and collect synthetic data to further train the models.

evaluated based on its ability to write complete functions and pass unit tests. However, we argue that an AI assistant's performance on coding benchmarks does not necessarily translate to its ability to support programmers. In practice, since a programmer mainly accepts small snippets of code suggestions from the AI assistant, it is not necessary for the AI assistant to be able to fully complete the entire task on its own. As such, the ideal evaluation of an AI assistant's utility would be to conduct user studies where some programmers are provided with AI assistance and compared to those who are not.

While initial user studies have already demonstrated positive signals in AI assistants' ability to improve programmer productivity [18], evaluation of AI assistants is still generally challenging. Beyond the typical barriers to conducting user studies (e.g., time, resources, accessibility to users), the iterative nature of generative tasks like coding adds complexity to the evaluation. Additionally, collecting an extensive set of user data or preferences on the AI assistant to further improve or customize aspects of the AI assistant may be difficult. To alleviate these challenges, we propose a simulation environment that captures the interaction between a programmer and an AI assistant.

Simulation has been commonly used to evaluate and train AI agents in robotics [6] and transferring from simulation to the real world [9] is a common strategy. More recently, powerful LLMs have been used to simulate human preference and behavior [7, 17, 11]. Inspired by the recent literature, we rely on LLMs (specifically GPT-3.5) to simulate the programmer in our simulation environment. The challenge of creating a realistic simulation of human-AI interactions is appropriately modeling the programmer's behavior. We integrate prompting techniques with the existing literature that aims to understand how programmers use LLM assistance [13] to produce more nuanced behavior of how programmers write code and incorporate the AI assistant's suggestions, as illustrated in Figure 1.

Our simulation for a widely applicable use case like programming with an AI assistant can be useful for multiple reasons; we discuss two and provide associated experiments for each:

- **Efficient evaluation signal.** The environment can be used to more cheaply evaluate existing models to understand the nuances of supporting humans on tasks like programming. We compare different LLM models of varying abilities and show the relation between their coding abilities and their simulation performance.
- **Source of realistic, synthetic data.** Collection of human data may be augmented by data collected from our environment. We demonstrate the simulation environment's ability to be used as a source of data by collecting preferences of AI assistant suggestions.

2 Related Work

General Background. The emergence of artificially intelligent assistants based on large language models (LLMs), such as ChatGPT [16] and Copilot [8], holds the potential to enhance productivity in various industries. Studies have demonstrated significant productivity improvements through AI-assistants, with experiments showing a 14% increase in customer support agent productivity [4] and a randomized control trial revealing a 55% reduction in coding time with Copilot [18]. The backend of these systems is LLMs such as GPT which are trained on standard language modeling objectives using the Common Crawl data [19]. Chat models such as ChatGPT are fine-tuned using Reinforcement Learning from Human Feedback (RLHF) [22, 3] to be more conversational. Autocomplete models like Copilot are then fine-tuned on public code repositories [5], without any use of human feedback.

LLMs to simulate Humans. Motivated by the high cost of obtaining human data and running user studies, there is an increasing number of studies relying on LLMs to simulate human behavior. In [17], LLMs "generative agents" simulate believable human behavior in real life in a small city environment. Recent work on learning from "AI feedback" (RLAIF), as opposed to learning from human feedback, has used LLMs in order to simulate human preference for the training for chat models and has shown that AI preference is highly correlated with human preference [7]. Moreover, models trained using RLAIF are competitive with models trained using RLHF [11]. In particular, when training a smaller LLM with feedback from a larger LLM this strategy is particularly effective and can be considered as a form of distillation.

Coding Benchmarks. As mentioned previously, HumanEval [5] and MBPP [2] are the most commonly used coding benchmarks. However, there exist variations of these benchmarks to evaluate the fill-in-the-middle abilities of the models and single-line completion. The DS-1000 [10] benchmark evaluates models' abilities on complex data science problems that require using outside libraries. An example of more involved evaluations is the multi-turn program evaluation benchmark (MTPB) [14] which factorizes a coding task into a set of multiple prompts. Our simulation environment can be considered as accomplishing this factorizing synthetically with the code of the simulated programmer.

3 Simulation Environment of Programmer-LLM Interaction

We propose a simulation environment that captures the interaction between the programmer and the AI assistant. Among various forms of AI assistance in programming,[3] we consider a common setting of programming with an AI assistant (typically a LLM) that provides inline code suggestions inside the programmer's integrated development environment. In this environment, not only is the AI assistant powered by an LLM but the programmer is also simulated using a secondary LLM. While, in our experiments, we use variants of GPT-3.5 [15] to simulate the programmer and the open-source LLM CodeLlama 7B [21] as the AI assistant, both the programmer and AI assistant can be instantiated by any LLM that has the ability to perform code completion and code infilling. The code is restricted to Python only.

3.1 Overview of environment set-up

Task definition. The simulation requires that a coding task T is defined. The task is typically one that would require the programmer multiple actions (e.g., multiple lines of code) to complete. For example, a task could be to write a machine learning algorithm to classify cats and dogs. The

[3]A programmer may also "chat" with an LLM model by writing comments in their code files.

programmer is typically aware of the full task specifications, but the AI assistant is not. As such, in our environment, the programmer is provided with a natural language description of a coding task and a function signature, while the LLM only observes the code file.

Iterative interaction protocol. We now provide a high-level sketch that formalizes the interaction between the programmer and AI assistant to complete the task T:

1. Assume that the programmer performs the task in continuous time and that the state of the current code file that the programmer is editing at any time t is X_t.
2. At some time t, the AI assistant can generate a code suggestion S_t which is a function of the current code file: $S_t = \pi_{\text{AI}}(X_t; \theta_{\text{AI}})$. The suggestion is typically based on a prompt P_t extracted from the code file, rather than the entire code file as it might not fit in the model context size.
3. In response to the suggestion, the programmer has two choices: accept the suggestion or reject the suggestion. If the programmer accepts the suggestion at a time t', the code file is modified to include the suggestion and becomes $X_{t'}$. The programmer continues from this point can arbitrarily modify the code file.

Evaluation metrics. The objective of the programmer is to complete the task T. We assume that, at the end time t^e, the programmer will stop their session and the code they've written X_{t^e} is their attempt at completing the task T. Given their coding session and final code, we measure as metrics the time it took to complete the task $t^e \in \mathbb{R}$ and a binary success metric (pass@1) for whether the code accomplishes the task.

3.2 Simulating programmer behavior

In Section 3.1, we outlined the iterative interaction process between the programmer and the LLM but did not describe how the human behaves when writing code in Step 1 and Step 3. To ensure the simulation faithfully reflects real-world programmers, we design the programmer behavior in each step based on recent work that has performed a detailed analysis of the patterns of behavior of programmers writing code with GitHub's Copilot [13].[4] All activities are implemented with a custom prompt template provided to either the chat GPT-3.5 ("gpt-3.5-turbo") or the base GPT-3.5 ("gpt-3.5-turbo-instruct"). Each of these nine activities entails different behavior from the programmer and is implemented with a custom prompt template.

The simulation consists of an iterative process where in Step 1 the programmer takes a writing action randomly assigned with probabilities according to the transitions in the Markov Process described in [13], then in Step 2, the model generates a suggestion, and finally in Step 3 the programmer takes a thinking action also randomly assigned accordingly to the probabilities in [13] and then edits the code if needed.

More specifically, when the programmer is writing code (Step 1), there are a few states that we randomly assign the programmer according to a probability distribution of states determined from [13]:

- `Writing new code functionality`: we prompt GPT-3.5 base from the existing code to write $n \sim Poisson(10)$ tokens. This takes a time equal to the number of tokens written. We put a negative logit bias of -100 on all tokens that contain "#" to prevent comments in this activity.
- `Prompt crafting for suggestions`: we prompt GPT-3.5 base to complete the code starting from the comment "# write a comment in English and not code to describe what the next step in the code should be <newline> # the next step is". This takes a time equal to the number of tokens written.
- `Writing documentation`: we prompt GPT-3.5 base to complete the code starting from the comment "# write a comment in English to describe what the code is doing <newline> # the code above"
- `Editing code`: We prompt GPT-3.5 to re-write the entire code given the old code with a similar number of tokens $\pm$ 5 tokens. This takes a time equal to the edit distance between the original code and the edited code

[4]Note that Mozannar et al. [13] has two additional activities, testing code and looking up documentation, which we leave to future work.

Preamble	A good suggestion is one that helps a programmer accomplish the task without error given the existing code body. If we append the suggestion to the code body, it should be better than only the code body. We define four evaluation axes for suggestion quality: helpfulness, accuracy, conciseness, and readability. We define helpfulness as the ability of the suggestion to help the programmer complete the task from the existing code We define accuracy as the ability of the suggestion to help the programmer accomplish the task without error, the suggestion can be incomplete. We define conciseness as the ability of the suggestion to help the programmer accomplish the task with the least amount of code and using existing libraries and functions. We define readability as the ability of the suggestion to help the programmer accomplish the task with the most readable code. You are an expert Python programmer writing code with the help of an assistant who gives you partial suggestions. Given a task, a code that is already written and a code suggestion, you have to output 1 if is a good suggestion to complete the code for the task, and 0 otherwise .
1-Shot Exemplar	—– Example —– Task: Given a list of integers, return the sum of all the integers in the list. Code body: def sum_list(list):<newline><tab>sum = 0 <newline><tab><tab> for i in list: Suggestion: <tab><tab> sum += i <newline><tab> return sum Thoughts on Suggestion: Helpfulness - 10. The suggestion completes where the code left off and completes the task. Accuracy - 10. The suggestion has no errors and gets the sum of the integers in the list Conciseness - 5. The code can be made more concise by using the built-in sum function Readability - 10. The code is readable and easy to understand <newline> good suggestion=1
Sample to annotate	Follow the instructions and the example above and output 1 if it is a good suggestion and 0 otherwise. Task: [**Task Description**] Code body: [**Code Body**] Suggestion: [**Suggestion**]
Ending	good suggestion=

Table 1: Prompt used for determining whether the simulated programmer should accept a given suggestion in the context of completing a task with existing code. This mimics the prompt template in [11].

- `Waiting for a suggestion`: The simulated programmer writes no tokens.

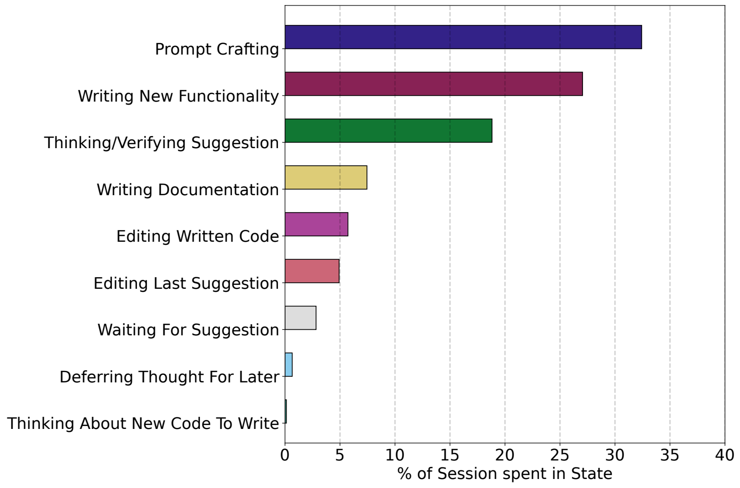
After the writing action, a suggestion is generated by the LLM. When the programmer is reviewing and potentially incorporating the suggested code (Step 3), we again assign them randomly to one of the following states:

- `Verifying the suggestion`: we implement the decision to accept a suggestion with a chain-of-thought (COT) with one example (COT 1-shot) and self-consistency following [11] to GPT-3.5-turbo, the prompt is showcased in Table 1. This takes a time equal to half the tokens in the suggestion.
- `Deferring thought`: we automatically accept the suggestion. This takes a time equal to 1/10 of the number of tokens in the suggestion.
- `Thinking about new code to write`: we automatically reject the suggestion
- `Edit the suggestion`: we implement the decision to edit a suggestion with a COT 1-shot prompt. This takes time equal to half the tokens in the suggestion (to verify) plus the edit distance between the original and edited suggestion.

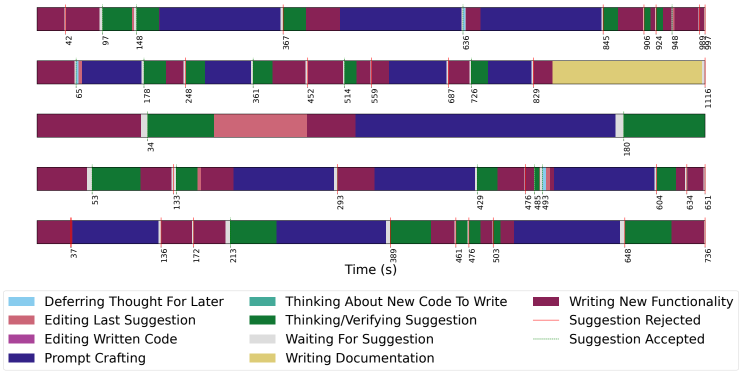
The simulation ends once the simulated programmer starts writing code outside the scope of the function or we reach 10 iterations.

3.3 Telemetry

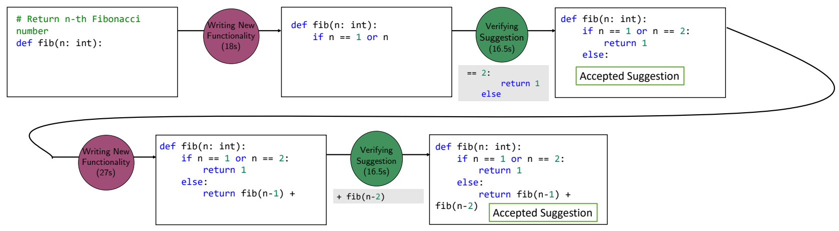
The simulation also logs aspects of the interactions via *telemetry* that mimics the real-world telemetry of interacting with LLMs. Before and after each programmer activity, we log the code file, the suggestion if it exists, the programmer actions (accept/reject), edits to the suggestion, and time. The telemetry can be represented as a dataset $D = \{c_i, S_i, A_i, X_i, t_i, z_i\}$ where c_i is the activity name, S_i is the suggestion, A_i is the action, X_i is the code, t_i is time and z_i represents auxiliary information such as information about suggestion edits (if it exists).



(a) The percentage of total session time spent in each activity during a coding session.



(b) Timelines annotated with coding activities, suggestion acceptance/rejection for 5 randomly selected tasks in the simulation.



```
# Return n-th Fibonacci
number
def fib(n: int):
```

Writing New Functionality (18s)

```
def fib(n: int):
    if n == 1 or n
```

Verifying Suggestion (16.5s)

```
== 2:
        return 1
    else
```

```
def fib(n: int):
    if n == 1 or n == 2:
        return 1
    else:
```

Accepted Suggestion
Writing New Functionality (27s)

```
def fib(n: int):
    if n == 1 or n == 2:
        return 1
    else:
        return fib(n-1) +
```

Verifying Suggestion (16.5s)

```
+ fib(n-2)
```

```
def fib(n: int):
    if n == 1 or n == 2:
        return 1
    else:
        return fib(n-1) +
fib(n-2)
```

(c) Code snippets from the simulation obtain the n'th Fibonacci number, the snippets illustrate every step of the iteration annotated with the code activities and time spent in each activity. Both suggestions displayed were accepted in this example. These snippets are obtained from the simulation without code edits.

Figure 2: Illustration of aggregate programmer behavior in the simulation as well as snippets of the coding simulation.

4 Evaluation

Validation of Simulation Environment. To validate programmer and AI assistant interactions in the simulation, we use the HumanEval dataset which consists of 164 Python problems each with an associated docstring, a ground truth function body solution, and a set of tests to evaluate the validity of the proposed solutions [5]. We run the simulation for each example in the HumanEval and record the telemetry for each example. In Figure 2 (a) we show the distribution of time spent in the nine different activities across all 164 tasks and Figure 2 (b) shows a timeline of the coding snippet as transitions between the nine activities. The three most common activities in the simulation are writing code by the simulated programmer, prompt crafting, and verifying suggestions of the AI assistant, which aligns with the study of real programmer behavior where they found that the four most common states are verifying suggestions, writing code, and thinking about code to write. In Figure 2 (c), we can see how the AI assistant provided short code snippets that helped the programmer complete the task.

Performance of Different Models in Simulation. Fixing the simulated programmer to be GPT-3.5, we vary the LLM model between three models of increasing size: CodeLlama 7B [21], CodeLlama 13B [21] and GPT-3.5-turbo-instruct. We set a maximum token constraint of 20 for each of the models. We also disable the action of "Editing Code" as it forces the code to continue for too many rounds. The coding ability of these three models on HumanEval is reported in Table 3 according to the literature. We plug the models in the simulation and report the average metrics on the 164 problems of HumanEval in Table 2. The three metrics we report are pass1 (did code pass the unit tests?), average time spent in simulation, and finally acceptance rate by the human proxy. We can see that all three metrics improve with the performance of the LLM model. Moreover, we can see that the performance of the programmer-LLM pairing lies between the performance of the LLM and the programmer proxy (GPT-3.5). However, we also see that the lower quality LLMs (CodeLlama 7B and 13B) can bring down the performance of the programmer.

Table 2: Comparison of Different Model Configurations

Metrics	GPT-3.5 + CodeLlama-7B	GPT-3.5 + CodeLlama-13B	GPT-3.5 + GPT-3.5
pass@1 (%)	36.1	43.6	49.4
average time	413	401	344
acceptance rate (%)	24.2	39.9	53.0

Table 3: Performance of the models on HumanEval dataset in terms of pass@1 as reported in the literature [21].

Model	**CodeLlama 7B**	**CodeLlama 13B**	**GPT-3.5**
Pass@1	33.5%	36%	48.1%

Modifying the LLM based on the Simulation. The simulation environment can be used not only to evaluate existing AI assistants but also to collect programmer preferences to modify or improve the AI assistant. We experiment with modifying the **length of the suggestions** that the LLM assistant provides. In particular, we constrain the length of the generation to be exactly n tokens and vary $n \in \{5, 10, 20, 30\}$ and run the simulation on HumanEval. Results are shown in Table 4. We observe that as we increase the suggestion length, the acceptance rates increases. However, as we increase the suggestion length the average time significantly increases. It is clear here that we should favor a smaller max token length of generation, as the performance of the programmer-LLM team decreases with larger suggestion sizes. It would be interesting to explore if the programmer proxy was a less capable model than the LLM, if we see a reverse trend.

Table 4: Comparison of different max token generation length for CodeLlama 7B on a sample of 30 HumanEval problems

Metrics Max Token Size	5	10	20	30
pass@1 (%)	50	43	33.3	43
average time	487.8	540	587	658
acceptance rate (%)	21.0	34	29.7	39.8

5 Limitation, Discussion and Conclusion

Limitations. Our simulation environment has many limitations. First of all, it is only designed for writing functions and it is mono-lingual (Python only for now). Second the cursor of the programmer is not taken into account, programmers often go back and forth in the code file to edit code that we do not model in detail, this is something that we plan to include in future iterations by keeping track of a dynamically changing cursor which would also allow to evaluate infilling abilities. Third, the simulation requires the use of a closed-source LLM model that is powerful enough to provide a good signal such as GPT-3.5, it would be interesting to see if we can replace GPT-3.5 with an open-source model and maintain a good simulation. Finally, we did not provide quantitative or qualitative measures to evaluate how realistic our environment is. We hope to remedy these limitations in future work.

Future Work. The hope is that this simulation provides an environment where we can train and evaluate LLMs with the explicit goal of making them better in-line assistants to humans. We hope to explore different strategies that exploit the telemetry generated by the simulation to train better LLM models and then conduct an evaluation with real human participants to check if simulation gains translate to the real world.

References

[1] Amazon. Ml-powered coding companion – amazon codewhisperer, 2022. URL https://aws.amazon.com/codewhisperer/.

[2] 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.

[3] Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, et al. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*, 2022.

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

[5] 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.

[6] HeeSun Choi, Cindy Crump, Christian Duriez, Asher Elmquist, Gregory Hager, David Han, Frank Hearl, Jessica Hodgins, Abhinandan Jain, Frederick Leve, et al. On the use of simulation in robotics: Opportunities, challenges, and suggestions for moving forward. *Proceedings of the National Academy of Sciences*, 118(1):e1907856118, 2021.

[7] Yann Dubois, Xuechen Li, Rohan Taori, Tianyi Zhang, Ishaan Gulrajani, Jimmy Ba, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. Alpacafarm: A simulation framework for methods that learn from human feedback. *arXiv preprint arXiv:2305.14387*, 2023.

[8] Github. Github copilot - your ai pair programmer, 2022. URL https://github.com/features/copilot.

[9] Abhishek Kadian, Joanne Truong, Aaron Gokaslan, Alexander Clegg, Erik Wijmans, Stefan Lee, Manolis Savva, Sonia Chernova, and Dhruv Batra. Sim2real predictivity: Does evaluation in simulation predict real-world performance? *IEEE Robotics and Automation Letters*, 5(4): 6670–6677, 2020.

[10] 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.

[11] Harrison Lee, Samrat Phatale, Hassan Mansoor, Kellie Lu, Thomas Mesnard, Colton Bishop, Victor Carbune, and Abhinav Rastogi. Rlaif: Scaling reinforcement learning from human feedback with ai feedback. *arXiv preprint arXiv:2309.00267*, 2023.

[12] 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.

[13] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. Reading between the lines: Modeling user behavior and costs in ai-assisted programming. *arXiv preprint arXiv:2210.14306*, 2022.

[14] 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. *arXiv preprint arXiv:2203.13474*, 2022.

[15] OpenAI. Chatgpt: Optimizing language models for dialogue, 2022. URL `https://openai.com/blog/chatgpt/`.

[16] OpenAI. Chatgpt: Introducing chatgpt. `https://openai.com/blog/chatgpt`, 2022.

[17] Joon Sung Park, Joseph C O'Brien, Carrie J Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. *arXiv preprint arXiv:2304.03442*, 2023.

[18] 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.

[19] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[20] replit. Meet ghostwriter, your partner in code., 2023. URL `https://replit.com/site/ghostwriter`.

[21] 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.

[22] Daniel M Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. Fine-tuning language models from human preferences. *arXiv preprint arXiv:1909.08593*, 2019.