Large Language Models Should Ask Clarifying Questions to Increase Confidence in Generated Code*

Jie JW Wu
George Washington University
jiewu@gwu.edu

ABSTRACT

Large language models (LLMs) have significantly improved the ability to perform tasks in the field of code generation. However, there is still a gap between LLMs being capable coders and being top-tier software engineers. Based on the observation that top-level software engineers often ask clarifying questions to reduce ambiguity in both requirements and coding solutions, I argue that the same should be applied to LLMs for code generation tasks. By asking probing questions in various topics before generating the final code, the challenges of programming with LLMs, such as unclear intent specification, lack of computational thinking, and undesired code quality, may be alleviated. This, in turn, increases confidence in the generated code. In this work, I explore how to leverage better communication skills to achieve greater confidence in generated code. I propose a communication-centered process that uses an LLM-generated communicator to identify issues with high ambiguity or low confidence in problem descriptions and generated code. I then ask clarifying questions to obtain responses from users for refining the code.

CCS CONCEPTS

• **Software and its engineering** → **Designing software**; • **Computing methodologies** → **Generative and developmental approaches**.

KEYWORDS

Software Development, Large Language Models, Code Generation

> "Asking a good question can be valuable in and of itself, irrespective of the answer. It communicates your respect for the other person."
> - Adapted from the Iowa Peace Institute Message

1 INTRODUCTION

Large language models (LLMs) [9, 33, 36, 38], such as OpenAI's Codex [7], AlphaCode [17], and CodeGen [24], possess significant capabilities to generate code snippets from natural language requirements. However, there are several reported issues with LLMs, including problems with intent specification, problem decomposition [29], code quality, and overconfidence [19, 20], as well as usability [18]. These issues indicate that there is still a substantial gap between an LLM as a programming assistant [1, 6, 28, 34, 44] and a software engineer. As the responsibility of software developers encompasses more than just writing code, current LLMs cannot fully replace professional software developers [4, 29]. At a high level, the gap lies in several critical aspects of software development beyond coding, such as effective communications, requirements, design, domain knowledge, and the broader context of relevant projects and components, etc [23, 31, 32, 35]. In this paper, I am interested in applying the communication lens to inspect the gap, given that effective communication is a critical capability that connects all of the above-mentioned parts to coding. I study the following research question: *Does asking clarifying questions increase confidence in ChatGPT-generated code?*

With this question, let us take a step back to compare the communications of LLMs and software developers. In the literature, the level of communication skills is rarely emphasized or evaluated in the field of code generation. The current LLMs are evaluated by generating code in one or multiple attempts from one-off problem descriptions, without further conversational inputs [2, 7, 17]. This means when the input problem description is error-prone or incomplete without full context, the model has to generate the code without the chance to clarify questions that are necessary to ensure the quality and correctness of the code. On the contrary, given a software engineering task in real-world enterprises, professional developers use various ways of communication, such as asking more questions in 1:1 conversations, group meetings, and Slack channels to obtain more information and reduce ambiguity about the detailed requirements, context of the projects, and the design alternatives. Proactive and effective communication is a critical skill in practice for top-level software developers to accomplish their software engineering tasks reliably with high quality [14, 21, 22, 27, 40].

Inspired by this behavior, in this work, I would like to study the potential of LLMs from the dimension of effective communication skills. As the first step toward this objective, I explore a process centered around promoting effective communication to ask more clarifying questions to refine the final generated code from LLMs. The initial exploration via an empirical example indicates that the communication-centered process is promising in boosting the communication skills of LLMs for code generation tasks. Finally, I provide conclusions and future work.

2 COMMUNICATION-CENTERED PROCESS

In this section, I describe the proposed communication-centered process for code generation tasks. The communication-centered process ultilizes two components - a *coder* and a *communicator*. The visual illustration of the process is shown in Figure 1. The coder takes the problem description and, optionally, the information provided by the communicator to generate the code. The communicator review the problem descriptions and the code generated from previous iteration, then ask questions on whether certain parts are identified as low confidence and need to be clarified or probed via questions. Finally, the communicator sends back the responses from

---

*This paper is accepted and presented at the 7th Annual Symposium on Machine Programming (MAPS '23), held on December 3, 2023 in San Francisco, CA, USA.

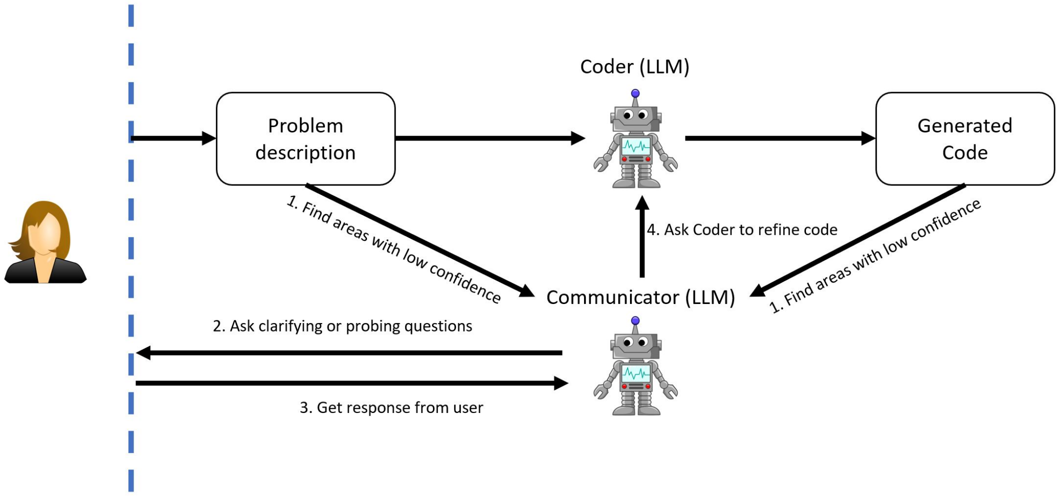

Coder (LLM)
Problem description
Generated Code
1. Find areas with low confidence
4. Ask Coder to refine code
1. Find areas with low confidence
Communicator (LLM)
2. Ask clarifying or probing questions
3. Get response from user

Figure 1: The visual illustration of the communication-centered process: the coder generates code according to the problem description and the communicator's information. The communicator detects the parts with low confidence where communication may help to get more information from the user. Then the communicator asks clarifying questions and gets the response. Finally, the response is sent back to the coder to refine the generated code.

users to the coder to refine the generated code. This process is repeated until some condition is met. Each of these two components can be implemented using a LLM.

Inspired by how the high-quality code is produced by top-level software developers, in the LLM-based code generation, my main idea for the communicator is that it should find the parts from the code and description that are typically of higher uncertainty and low confidence. These parts can be improved by getting more useful information from users via good clarifying and probing questions. The communicator asks questions focused on various aspects of software development concerning the high standards of final output code, such as intent specification, intent clarification, requirement disambiguity, coding style etc. I use an example in the next section to illustrate how the coder works with the communicator in detail.

3 EXAMPLE USAGE

I showcase a practical example using the communication-centered process for code generation tasks. In this example, I use ChatGPT 3.5 as the LLM for the coder and the communicator. As shown in Figure 2, a user is looking for a code snippet to return n-th Fibonacci number. Within the process, the coder LLM first generated an initial code snippet. From the initial code output that, although ChatGPT 3.5 as the coder generated the code as required, several topics on this can be further elaborated and discussed. In this case, the communicator analyzed the problem description and the initial code to generate a list of clarifying questions. The top question is concerned with the negative input, which was not mentioned in the problem description. The user responded the question to set the expectation on the case of the negative input. Then, the coder refined the code with that additional information. I only use one iteration in the example, but in theory this loop can continue until some criteria is met.

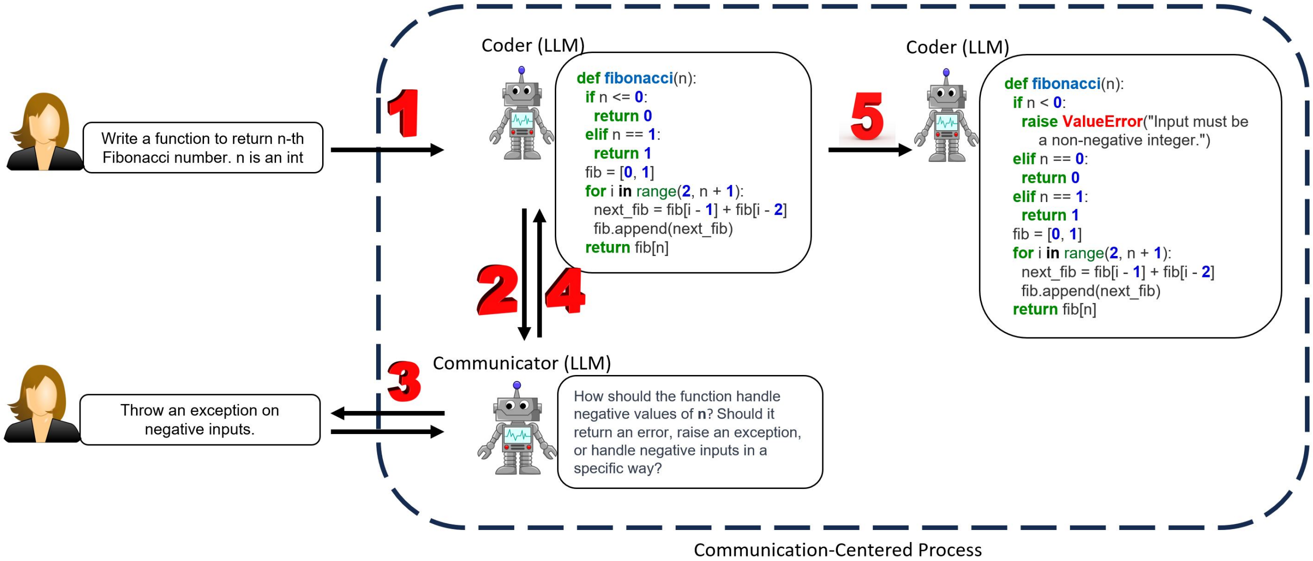
The key part of the process is how the communicator interacts with the user and the coder. For the communicator, I use ChatGPT 3.5 as the LLM, which is same as the coder. The communicator uses zero-shot prompting to generate the clarifying questions from the problem description and the code from the previous iteration. The example of prompt from Figure 2 is described below.

```
You are an expert in software engineering. You will be given
the problem description and current code of a coding task. You
will generate a list of clarifying questions that may result in
refining the code.

### Problem Description
"write a function to return n-th Fibonacci number. n is an int "

### Generated Code From Previous Iteration
def fibonacci(n):
  if n <= 0:
    return 0
  elif n == 1:
    return 1
  fib = [0, 1]
  for i in range(2, n + 1):
    next_fib = fib[i - 1] + fib[i - 2]
    fib.append(next_fib)
  return fib[n]
```

The output list of clarifying questions for the communicator in the above example is shown in Table 1. With the simple zero-shot prompt, the communicator is already able to automatically cover a number of topics that deserve asking questions. As the future



Write a function to return n-th Fibonacci number. n is an int

```
def fibonacci(n):
  if n <= 0:
    return 0
  elif n == 1:
    return 1
  fib = [0, 1]
  for i in range(2, n + 1):
    next_fib = fib[i - 1] + fib[i - 2]
    fib.append(next_fib)
  return fib[n]
```

How should the function handle negative values of **n**? Should it return an error, raise an exception, or handle negative inputs in a specific way?

Throw an exception on negative inputs.

```
def fibonacci(n):
  if n < 0:
    raise ValueError("Input must be
      a non-negative integer.")
  elif n == 0:
    return 0
  elif n == 1:
    return 1
  fib = [0, 1]
  for i in range(2, n + 1):
    next_fib = fib[i - 1] + fib[i - 2]
    fib.append(next_fib)
  return fib[n]
```

Figure 2: An example of the communication-centered process.

extension, one could modify the prompt to focus on topics that are more helpful for their customized usage or for the quality of generated code. For example, topics such as code context, coding style, non-functional requirements, code analysis, explanation are mentioned as the current weakness of the code output of LLMs [18]. So, these topics can be better prioritized in the prompt of the communicator. From this mere example, I see big potential that boosting communication skills can lead to improvements of the final code output in several topics such as negative input, documentation, etc.

4 BACKGROUND AND RELATED WORK

Code Generation with Large Language Models. In recent years, the field of code generation has seen a significant shift with the large language models. For example, Codex [7], fine-tuned on GPT-3 [5] on a large corpus of source code data, is capable of generating code for 47/164 problems in the HumanEval dataset in single run, a benchmark for code generation task. Codex became the core model for the Copilot [46], an AI-powered coding assistant developed by GitHub. After Codex, a couple of models similar to Codex but with smaller size were then developed, including GPT-J [37], CodeParrot [8], PolyCoder [42]. AlphaCode [17], with size comparable to Codex, was trained on Github data and fine-tuned on competition-level programing problems. It exceeded half of the competitors in coding competitions of CodeForces, a well-known online competitive programming platform. CodeGen [24] was trained on both natural language and programming language data for code generations with multi-turn prompts. However, the level of communication skills of these models are not emphasized and evaluated. These models are evaluated by generating code in one or multiple attempts from one-off problem descriptions, without further information from conversations. Therefore, when the input problem description is error-prone or incomplete, the model still has to generate the code without the chance to clarify critical questions. My work serves as an exploration to address this usability problem.

Self-Correct Large Language Models. Recently, a promising approach to improve the output efficiency of large language models is *self-correction* [26]. In self-correction approach, the LLM uses the feedback guided or prompted by itself to refine their results. One popular category of work uses human feedback to refine their results directly [10, 11, 16, 25, 30]. Other studies employed different strageties to self-correct LLMs using automated feedback such as self-training [3, 13], generate-then-rank [12, 39], feedback-guided decoding [41, 43], iterative post-hoc revision [15, 45], etc. My work in this paper is also under the category of self-correction using both human feedback and automated feedback, but with a new perspective of improving communication skills for code generation tasks.

5 CONCLUSION AND FUTURE WORK

In this exploratory paper, I showed an initial step on the problem of how to increase the communication skills of LLMs to elevate the final generated code via clarifying and probing questions. I argue that the proficiency of communication skills of LLMs is necessary for generating code with high standards. Elevated communication skills should be viewed as an important factor toward bridging the gap between LLMs and top-notch software developers. Specifically, this effort will not only increase confidence of generated code, but also gain the trust of users using them as a programmin assistant. Although it needs additional efforts of conversational input from users, I believe it is still necessary and worthwhile. As a first step toward this effort, I explored a communication-centered process that ultilizes a coder LLM and a communicator LLM that work together to ask clarifying questions and refine the generated code.

Topic	Clarifying Questions Generated by the Communicator LLM
Input Validation	Should input validation be part of the function?
	If so, what should be done if n is not an integer?
Error Handling	How should errors be handled if n is not a valid integer or if it's negative?
	Should the function raise exceptions, return error codes, or follow a different error-handling strategy?
Performance Requirements	Are there any performance requirements or constraints for this function?
	Is there a need for optimizing the function for efficiency, especially for large values of n?
Function Signature	Are there any specific naming conventions or requirements for the function name and parameters?
	Should the function take any additional parameters, options, or configuration settings?
Output Type	Is there a preferred return type for the function, such as an integer, a string, or another data type?
Default Values	Should the function provide default values for n if it's not provided by the caller?
Documentation	What documentation standards should be followed for this function, such as docstrings, comments, or external documentation?
Testing	Are there specific test cases or test scenarios that should be considered when testing the function?
	Should the function be tested for edge cases, such as very large values of n?
Language and Environment	Is there a specific programming language or environment in which this function should be implemented?
	Are there any platform or compatibility requirements?
Algorithm Choice	Is there a preference for using a specific algorithm for calculating Fibonacci numbers, such as an iterative or recursive approach?
	Should the function allow for different algorithm choices or optimizations?
Usability and Extensibility	Should the function be designed for easy integration into other codebases?
	Are there considerations for future extensions or modifications to the function?
Security Considerations	Are there any security considerations related to the input or output of the function that need to be addressed?
Concurrency and Parallelism	Does the function need to support concurrent or parallel execution for improved performance?
Version Control and Deployment	How will the function be versioned, deployed, and maintained in the software development lifecycle?
Use Case and Context	What is the primary use case or context in which this function will be used?
	Understanding the broader context can help in designing the function to meet specific needs.

Table 1: Example of clarifying questions in the Communicator LLM from Figure 2.

By generating clarifying questions in prompts via the communicator, there is hope to raise the capability of communication skills to produce code with higher confidence. However, several challenges still exist along this line of research and future work is needed, as described below.

Evaluation benchmark of communication skills for LLM-based code generation. This is needed to objectively quantify the capability of communication skills of LLM on code generation tasks and software engineering tasks. As aforementioned, the existing evaluation work of LLMs for code generation mainly focus on solving algorithm problems without additional converstational input. In my case, as future work, I will target at benchmark that the reveal how effective is the communcation ability of the model. For example, creating a dataset with blurred and noisy problem descriptions that hide some critical information is an interesting direction. In this setting, the model should ask the right questions rather than directly generate low-quality code.

Improving communication skills of LLMs. Besides benchmark, techniques to further improve the communication skills of LLMs can be the next steps as future work. Since the prompts are critical to the success of LLMs, one of the important future work is to design effective instructions w.r.t. communication skills. Another interesting angle is to study how to tune the model to switch between under-communicating, effective-communcating and over-communicating. I envision that different AI programming agents in future will have various levels and styles of communication ability. This work can be seen as the first step toward improving communication skills of LLMs for code.

6 ACKNOWLEDGEMENTS

I would like to thank the anonymous reviewers for their constructive feedback and suggestions. I also thank Vincent Hellendoorn for his insightful feedback after presenting this work at MAPS 2023.

REFERENCES

[1] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. 2019. Code2Vec: Learning Distributed Representations of Code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

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

[3] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, Carol Chen, Catherine Olsson, Christopher Olah, Danny Hernandez, Dawn Drain, Deep Ganguli, Dustin Li, Eli Tran-Johnson, Ethan Perez, Jamie Kerr, Jared Mueller, Jeffrey Ladish, Joshua Landau, Kamal Ndousse, Kamile Lukosiute, Liane Lovitt, Michael Sellitto, Nelson Elhage, Nicholas Schiefer, Noemí Mercado, Nova DasSarma, Robert Lasenby, Robin Larson, Sam Ringer, Scott Johnston, Shauna Kravec, Sheer El Showk, Stanislav Fort, Tamera Lanham, Timothy Telleen-Lawton, Tom Conerly, Tom Henighan, Tristan Hume, Samuel R. Bowman, Zac Hatfield-Dodds, Ben Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom Brown, and Jared Kaplan. 2022. Constitutional AI: Harmlessness from AI Feedback. *CoRR* abs/2212.08073 (2022).

[4] Ali Borji. 2023. A categorical archive of chatgpt failures. *arXiv preprint arXiv:2302.03494* (2023).

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

[6] N. D. Bui, Y. Yu, and L. Jiang. 2021. InferCode: Self-supervised Learning of Code Representations by Predicting Subtrees. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1186–1197.

[7] 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. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[8] Hugging Face. 2023. *Codeparrot.* https://huggingface.co/codeparrot/codeparrot

[9] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, and D. Jiang et al. 2020. CodeBERT: A Pre-trained Model for Programming and Natural Languages. *arXiv preprint arXiv:2002.08155* (2020).

[10] Patrick Fernandes, Aman Madaan, Emmy Liu, António Farinhas, Pedro Henrique Martins, Amanda Bertsch, José G. C. de Souza, Shuyan Zhou, Tongshuang Wu, Graham Neubig, and André F. T. Martins. 2023. Bridging the Gap: A Survey on Integrating (Human) Feedback for Natural Language Generation. *CoRR* abs/2305.00955 (2023).

[11] Amelia Glaese, Nat McAleese, Maja Trebacz, John Aslanides, Vlad Firoiu, Timo Ewalds, Maribeth Rauh, Laura Weidinger, Martin J. Chadwick, Phoebe Thacker, Lucy Campbell-Gillingham, Jonathan Uesato, Po-Sen Huang, Ramona Comanescu, Fan Yang, Abigail See, Sumanth Dathathri, Rory Greig, Charlie Chen, Doug Fritz, Jaume Sanchez Elias, Richard Green, Sona Mokrá, Nicholas Fernando, Boxi Wu, Rachel Foley, Susannah Young, Iason Gabriel, William Isaac, John Mellor, Demis Hassabis, Koray Kavukcuoglu, Lisa Anne Hendricks, and Geoffrey Irving. 2022. Improving Alignment of Dialogue Agents via Targeted Human Judgements. *CoRR* abs/2209.14375 (2022).

[12] Hangfeng He, Hongming Zhang, and Dan Roth. 2023. Rethinking with Retrieval: Faithful Large Language Model Inference. *CoRR* abs/2301.00303 (2023).

[13] Jiaxin Huang, Shixiang Shane Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. 2022. Large Language Models Can Self-Improve. *CoRR* abs/2210.11610 (2022).

[14] Mehdi Jazayeri. 2004. The education of a software engineer. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.* IEEE, xviii–xxvii.

[15] Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. SelfEvolve: A Code Evolution Framework via Large Language Models. *CoRR* abs/2306.02907 (2023).

[16] Julia Kreutzer, Shahram Khadivi, Evgeny Matusov, and Stefan Riezler. 2018. Can Neural Machine Translation Be Improved with User Feedback?. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT).*

[17] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[18] Jenny T Liang, Chenyang Yang, and Brad A Myers. 2023. Understanding the Usability of AI Programming Assistants. *arXiv preprint arXiv:2303.17125* (2023).

[19] Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D Le, and David Lo. 2023. Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues. *arXiv preprint arXiv:2307.12596* (2023).

[20] Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. 2023. No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT. *arXiv preprint arXiv:2308.04838* (2023).

[21] Ian R McChesney and Seamus Gallagher. 2004. Communication and co-ordination practices in software engineering projects. *Information and Software Technology* 46, 7 (2004), 473–489.

[22] Ivan Mistrík, John Grundy, Andre Van der Hoek, and Jim Whitehead. 2010. *Collaborative software engineering: challenges and prospects.* Springer.

[23] N. Nguyen and S. Nadi. 2022. An Empirical Evaluation of GitHub Copilot's Code Suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories.* 1–5.

[24] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[25] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. 2022. Training Language Models to Follow Instructions with Human Feedback. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS).*

[26] Liangming Pan, Michael Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. 2023. Automatically Correcting Large Language Models: Surveying the landscape of diverse self-correction strategies. *arXiv preprint arXiv:2308.03188* (2023).

[27] Roger S Pressman. 2005. *Software engineering: a practitioner's approach.* Palgrave macmillan.

[28] M. Rabinovich, M. Stern, and D. Klein. 2017. Abstract Syntax Networks for Code Generation and Semantic Parsing. *arXiv preprint arXiv:1704.07535* (2017).

[29] Advait Sarkar, Andrew D Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. 2022. What is it like to program with artificial

intelligence? *arXiv preprint arXiv:2208.06213* (2022).

[30] Jérémy Scheurer, Jon Ander Campos, Tomasz Korbak, Jun Shern Chan, Angelica Chen, Kyunghyun Cho, and Ethan Perez. 2023. Training Language Models with Language Feedback at Scale. *CoRR* abs/2303.16755 (2023).

[31] M. L. Siddiq, S. H. Majumder, M. R. Mim, S. Jajodia, and J. C. Santos. 2022. An Empirical Study of Code Smells in Transformer-Based Code Generation Techniques. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 71–82.

[32] D. Sobania, M. Briesch, and F. Rothlauf. 2022. Choose Your Programming Copilot: A Comparison of the Program Synthesis Performance of GitHub Copilot and Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1019–1027.

[33] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan. 2020. Intellicode Compose: Code Generation Using Transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1433–1443.

[34] M. Tufano, D. Drain, A. Svyatkovskiy, S. Deng, and N. Sundaresan. 2020. Unit Test Case Generation with Transformers and Focal Context. *arXiv preprint arXiv: Software Engineering* (2020).

[35] P. Vaithilingam, T. Zhang, and E. L. Glassman. 2022. 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*. 1–7.

[36] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. 2017. Attention is All You Need. In *Advances in Neural Information Processing Systems*, Vol. 30.

[37] Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 billion parameter autoregressive language model.

[38] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. 2021. CodeT5: Identifier-Aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. *arXiv preprint arXiv:2109.00859* (2021).

[39] Yixuan Weng, Minjun Zhu, Fei Xia, Bin Li, Shizhu He, Kang Liu, and Jun Zhao. 2023. Large Language Models Are Better Reasoners with Self-Verification. *CoRR* abs/2212.09561 (2023).

[40] Jim Whitehead. 2007. Collaboration in software engineering: A roadmap. In *Future of Software Engineering (FOSE'07)*. IEEE, 214–225.

[41] Yuxi Xie, Kenji Kawaguchi, Yiran Zhao, Xu Zhao, MinYen Kan, Junxian He, and Qizhe Xie. 2023. Decomposition Enhances Reasoning via Self-Evaluation Guided Decoding. *CoRR* abs/2305.00633 (2023).

[42] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 1–10.

[43] Kaiyu Yang, Jia Deng, and Danqi Chen. 2022. Generating Natural Language Proofs with Verifier-Guided Search. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 89–105.

[44] W. Ye, R. Xie, J. Zhang, T. Hu, X. Wang, and S. Zhang. 2020. Leveraging Code Generation to Improve Code Retrieval and Summarization via Dual Learning. In *Proceedings of The Web Conference 2020*. 2309–2319.

[45] Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-Edit: Fault-Aware Code Editor for Code Generation. *CoRR* abs/2305.04087 (2023).

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