The Programmer's Assistant User Experience

Steven I. Ross
IBM Research AI
Cambridge, MA, USA
steven_ross@us.ibm.com

Stephanie Houde
IBM Research AI
Cambridge, MA, USA
Stephanie.Houde@ibm.com

Fernando Martinez
IBM Argentina
La Plata, Buenos Aires, Argentina
martferc@ar.ibm.com

Michael Muller
IBM Research AI
Cambridge, MA, USA
michael_muller@us.ibm.com

Justin D. Weisz
IBM Research AI
Yorktown Heights, NY, USA
jweisz@us.ibm.com

ABSTRACT

The Programmer's Assistant is an experimental prototype development environment that integrates a conversational chat component with a code editor. The conversational aspect is powered by a code-fluent large language model. The prototype user experience provides for discussion of code in the code editor as well as for easy transfer of generated code to the editor under human supervision.

CCS CONCEPTS

• **Human-centered computing** → **HCI theory, concepts and models**; • **Software and its engineering** → *Designing software*; • **Computing methodologies** → *Generative and developmental approaches.*

KEYWORDS

code-fluent foundation models, conversational interaction, human-centered AI

ACM Reference Format:
Steven I. Ross, Stephanie Houde, Fernando Martinez, Michael Muller, and Justin D. Weisz. 2023. The Programmer's Assistant User Experience. In *28th International Conference on Intelligent User Interfaces (IUI '23 Companion), March 27–31, 2023, Sydney, NSW, Australia.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3581754.3584143

1 INTRODUCTION

The Programmer's Assistant [6] prototype was developed to investigate the viability of a interactive conversational agent chatbot in a software development setting, in an attempt to approach the vision laid out by Rich and Waters for the Programmer's Apprentice [3], where they introduced the concept of an artificial collaborative partner that could help software engineers with writing code, designing software systems, and creating requirements specifications. Its conversational capabilities are based on the same Codex large language model [1] that powers Github Copilot [2]. Providing conversational access to this model enables access to knowledge and abilities embedded in the model that are not evident in a command-completion format. Unlike ordinary chatbots, this application permits an artifact-centric conversation that requires more than a simple chat interface in order to enable conversation about the program under development.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
IUI '23 Companion, March 27–31, 2023, Sydney, NSW, Australia
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0107-8/23/03.
https://doi.org/10.1145/3581754.3584143

2 OUR GOALS FOR THE PROGRAMMER'S ASSISTANT

We are interested in developing AI-based tools that enhance the productivity and efficiency of software developers. We want these developers to find it easier and more enjoyable to perform the kinds of tasks that they are already performing, to provide help when wanted and needed, and not to hinder their efforts when help is not needed. To pursue these goals, we have designed the assistant to be available on demand and not monitor the user's work in progress or give unsolicited suggestions or advice, in keeping with the conversational agent interaction model proposed in Ross et al. [4, 5]. We rely on the user to take initiative and only have the assistant respond to their requests. In this way, the assistant can provide help when requested without undesirable interruptions that can distract or interfere with the user's flow.

3 CONVERSATIONAL INTERACTION

While command-completion is a useful and valuable function, there are many other opportunities to assist the software developer for which it is not particularly well suited. Conversation allows for the developer to ask questions which may or may not have code as answers. It enables follow-up discussion, requests for explanation of, modifications to, or alternatives for existing code. It also allows for incremental specification, recognizing that it is difficult to completely and unambiguously describe desired functionality. With a conversational interface, the system could potentially request clarification or offer choices between alternative interpretations of a user request, or could make assumptions which the user could subsequently accept or correct.

We achieve conversational interaction in the Programmer's Assistant by prompting the large language model with a prompt that establishes a script-like pattern in which two characters, the user and the assistant, are participating in a dialog. Then we extend the prompt incrementally, by adding each conversational turn by the user to the script, and allowing the model to generate the agent's response. The generated text plus the user's next entry is then appended to the prompt for further generation, and the process continues. The conversational prompt grows over the course of

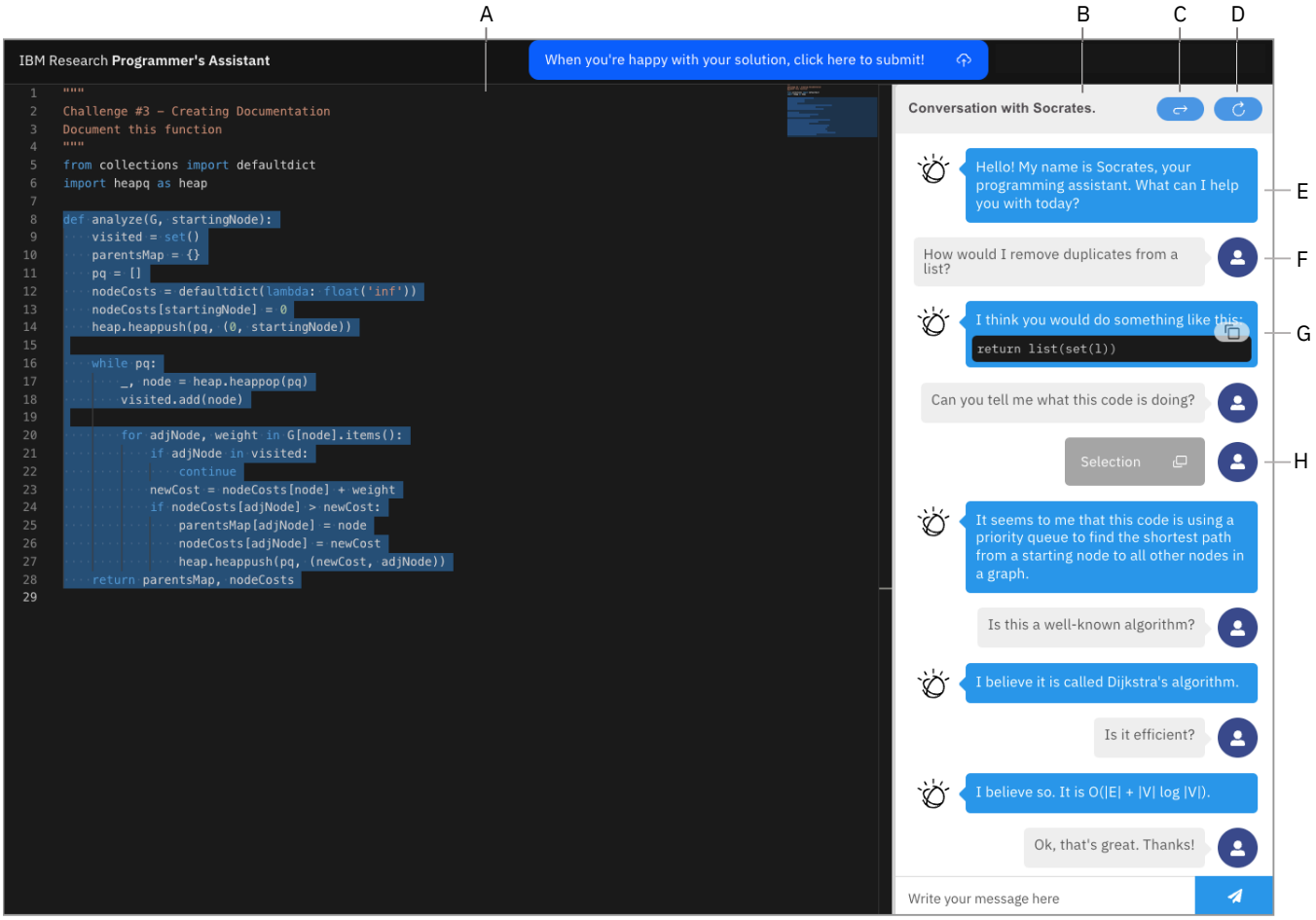


Figure 1: The Programmer's Assistant. The user interface provides a code editor on the left (A) and a chat pane on the right (B). The "try again" button (C) allows users to ask the assistant to generate an alternate response to the most recent question. The "start over" button (D) resets the conversational context for the assistant, but maintains the chat transcript in the UI. In this example, we show the assistant introduce itself to the user (E). Next, the user asks a general programming question (F), for which the assistant provides an inline code response (G). The user then asks a question about code selected in the editor (H), along with a series of follow-up questions.

the dialogue, providing context for future generation steps and providing a kind of short-term memory that allows the generation to be affected by past interactions in a session. Limitations on the maximum number of tokens that the model can handle require that the prompt cannot grow without bounds, and earlier interactions must be dropped in order to remain within the constraints imposed by the system. The initial prompt consists of a prologue which sets the stage for the conversation and establishes the persona of the assistant and interaction conventions. It also sets up a convention for how the assistant is to communicate code in a form that makes it easy for our client user interface to parse the code from model responses and display it with the proper programming language settings or incorporate code that the user selects in the editor into the conversation and treat it distinctly from textual entries or responses. The prologue is followed by several example interactions that reinforce the described assistant characteristics and illustrate the patterns of interaction that we wish to achieve.

4 ARTIFACT-GROUNDED CONVERSATION

A conversation is itself an artifact being collaboratively generated by the participants, but we are interested in situations where there is also another artifact of interest that the user is in the process of creating, modifying, or examining. In the case of software development, that artifact is often code, typically within some kind of integrated development environment. In our initial prototype, we provide a web-based code editor to serve as the artifact development environment, and provide a modestly sized, minimizable chat area on the side where conversations with the assistant can be conducted.

Some interactions with the user can take place entirely in the chat, such as when the user wants to ask a coding question such

as *"How do I get the keys in a python dictionary?"* or *"What's the best way to read the contents of a csv file?"* The results of such questions may be code or text or some combination of the two, and will appear in the chat. If a code response is just a few lines, it will appear inline in the chat, with a copy icon button to enable easy one-click copying of the code to the clipboard. Code responses longer than six lines appear in the chat in an iconified form and open in a separate pop-up window for readability. That window contains a button to allow the contents to be copied and the window closed in one click. The assistant never modifies the code in the code editor, but makes it easy to copy and paste code to the editor, leaving the user firmly in control.

Other conversational interactions are grounded in the context of code selected by the user in the code editor. Should a user wish to consult with the assistant on any text in the editor, they can simply select that text and type an entry in the chat. The selected text will be included in the conversation and will appear in the chat in iconified form. The user can thus make requests like *"What does this code do?"*, *"Translate this code to Java"*, *"Can you write a unit test for this function?"* or *"Document this"*. If the conversation continues with the selection remaining unchanged in the editor, subsequent exchanges will not redundantly include the selection in the chat.

In the current implementation, this is the only way that the programmer's assistant accesses the user's code. It does not otherwise monitor their work in the code editor or make unsolicited comments or suggestions.

5 DEALING WITH IMPERFECT RESPONSES

Despite the impressive capabilities of the Codex model, it sometimes provides incorrect responses or claims to not know the answer to a question that it actually is capable of answering. Given the probabilistic nature of the text generation process, resubmitting the question in the same context can often result in a different response, so we provide a *try-again* button that does just that, enabling a user to see alternative answers if the first one is deemed inadequate. Prior responses remain in the chat transcript, but are not included in the prompt so they will not influence future response generation.

While the accumulation of dialog context provides the assistant with a memory of the conversation and allows it to carry on a coherent multi-turn conversation, occasionally the accumulated context results in the assistant becoming stuck or fixated on some past part of the conversation and unable to continue the interaction productively. For such situations, we provide a *start-over* button that clears the conversational context while leaving the chat transcript intact. The assistant will forget the conversation up to this point, but will be able to move forward to respond to new requests.

6 THE PROGRAMMER'S ASSISTANT IN ACTION

A screen shot of the Programmer's Assistant prototype is presented in Figure 1. The sample conversation in the chat and code displayed in the editor provide a typical example of what a user sees when interacting with the system. In the depicted conversation, we show how the assistant introduced itself to the user. The user then asked a general programming question for which the assistant provided an inline code response. The user then implicitly changed the subject and went on to ask a question about code selected in the editor, for which they got an answer back from the assistant. After that, a series of follow up questions and answers between user and assistant ensued. The video demonstration that accompanies this paper depicts this kind of experience in motion.

7 CONCLUSION

Our original goal in creating the Programmer's Assistant was to develop a prototype user experience to test whether a conversational style of interaction would prove useful and acceptable to potential users of a software engineering assistance tool. We present the results of that user study in [6]. We assume that the rapid improvement in the quality of responses available from large language models will continue, but that imperfect results will always continue to be an issue due to imprecise communication and specification of desires, mismatched assumptions, and unstated or ill-formed goals. Nevertheless, we were surprised by the quality of results that were achievable with current technology and the usefulness of the resulting system for supporting software engineering activities.

The Programmer's Assistant has shown itself capable of providing a broad range of programming services to assist the user. It can answer questions, generate code upon request, consult about user selected code, translate between programming languages, and generate unit tests and documentation. These capabilities are a testament to the embedded knowledge and quality of the Codex large language model [1], but the conversational interface provided by the Programmer's Assistant makes them available to the user upon request. The ability to discuss generated code, asking for explanations, corrections, or alternative implementations in an extended multi-turn conversation results in a completely different co-creative experience than a command-completion interface such as that provided by Github Copilot [2]. These different approaches are not necessarily competing alternatives. Perhaps we will see them in an integrated form in a future software development environment.

REFERENCES

[1] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Arun Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating a Large Language Models Trained on Code.

[2] GitHub, Inc. 2022. *GitHub copilot · your AI pair programmer.* Retrieved August 5, 2022 from https://github.com/features/copilot/

[3] Charles H. Rich and Richard C. Waters. 1990. *The Programmer's Apprentice.* Addison-Wesley Publishing Company, Reading, MA.

[4] Steven Ross, Elizabeth Brownholtz, and Robert Armes. 2004. A Multiple-Application Conversational Agent. In *Proceedings of the 9th International Conference on Intelligent User Interfaces* (Funchal, Madeira, Portugal) *(IUI '04).* Association for Computing Machinery, New York, NY, USA, 319–321.

[5] Steven Ross, Elizabeth Brownholtz, and Robert Armes. 2004. Voice User Interface Principles for a Conversational Agent. In *Proceedings of the 9th International Conference on Intelligent User Interfaces* (Funchal, Madeira, Portugal) *(IUI '04).* Association for Computing Machinery, New York, NY, USA, 364–365.

[6] Steven I Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D Weisz. 2023. The Programmer's Assistant: Conversational Interaction with a Large Language Model for Software Development. In *28th International Conference on Intelligent User Interfaces.*