Experiences from Using Code Explanations Generated by Large Language Models in a Web Software Development E-Book

Stephen MacNeil
Temple University
Philadelphia, PA, USA
stephen.macneil@temple.edu

Andrew Tran
Temple University
Philadelphia, PA, USA
andrew.tran10@temple.edu

Arto Hellas
Aalto University
Espoo, Finland
arto.hellas@aalto.fi

Joanne Kim
Temple University
Philadelphia, PA, USA
joanne.kim@temple.edu

Sami Sarsa
Aalto University
Espoo, Finland
sami.sarsa@aalto.fi

Paul Denny
The University of Auckland
Auckland, New Zealand
paul@cs.auckland.ac.nz

Seth Bernstein
Temple University
Philadelphia, PA, USA
seth.bernstein@temple.edu

Juho Leinonen
Aalto University
Espoo, Finland
juho.2.leinonen@aalto.fi

ABSTRACT

Advances in natural language processing have resulted in large language models (LLMs) that can generate code and code explanations. In this paper, we report on our experiences generating multiple code explanation types using LLMs and integrating them into an interactive e-book on web software development. Three different types of explanations – a line-by-line explanation, a list of important concepts, and a high-level summary of the code – were created. Students could view explanations by clicking a button next to code snippets, which showed the explanation and asked about its utility. Our results show that all explanation types were viewed by students and that the majority of students perceived the code explanations as helpful to them. However, student engagement varied by code snippet complexity, explanation type, and code snippet length. Drawing on our experiences, we discuss future directions for integrating explanations generated by LLMs into CS classrooms.

CCS CONCEPTS

• **Social and professional topics** → *Computing education*; • **Computing methodologies** → *Natural language generation.*

KEYWORDS

large language models, explanations, computer science education

ACM Reference Format:
Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. 2023. Experiences from Using Code Explanations Generated by Large Language Models in a Web Software Development E-Book. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023), March 15–18, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3545945.3569785

SIGCSE '23, March 15–18, 2023, Toronto, ON, Canada.

ACM ISBN 978-1-4503-9431-4/23/03…$15.00
https://doi.org/10.1145/3545945.3569785

1 INTRODUCTION

Good explanations can help students learn introductory programming concepts [25]. In particular, explanations of code can assist understanding at multiple levels, from low-level explanations of particular lines of code and their syntax, up to more abstract explanations of the purpose of code fragments. Students could benefit greatly from the ability to access appropriate explanations at different levels of abstraction when interacting with learning resources. However, the workload required to generate large quantities of high-quality explanations for code snippets in learning resources can be a significant barrier for instructors. Previous work has explored automated methods to trace the execution of code [15, 29], define terms [17], give hints [35], and provide error-specific feedback [25, 35]. Most of these techniques require manual up-front effort by the instructor to generate hints, explanations, and feedback, that are later delivered to students when they need help.

Large language models (LLMs) provide new opportunities to support software engineering and development. For example, LLMs currently power tools like Github Copilot, Amazon Codewhisperer, and Tabnine which can generate code based on natural language specifications provided by a software developer [3, 6, 38]. For CS educators, LLMs present both opportunities and challenges. LLMs can be used by instructors to generate assignments more efficiently, saving valuable time [37]. Instructors can also use LLMs to automatically generate explanations for students in real-time [24, 37]. These generative models are very new, however, and it is not yet clear to what extent they can facilitate learning in practice.

1.1 Motivating example

Consider the following JavaScript program that uses the Deno runtime to launch a simple webserver, typical of an early program that might appear in a web development course.

```
import { serve } from "https://deno.land/std@0
    ↪ .140.0/http/server.ts";
let count = 0;
const handleRequest = (request) => {
  if (request.method === "POST") {
    count++;
  }
  return new Response(count);
};
serve(handleRequest, { port: 7777 });
```

Students viewing this code listing might benefit from different kinds of explanations depending on their level of knowledge. For example, a student with prior experience in web development may find a succinct summary of the purpose of the program valuable, whereas a student less familiar with JavaScript might prefer a line-by-line explanation that highlights aspects of syntax and terminology. Producing appropriate explanations up front that are suitable to the needs of a diverse cohort of learners is time-consuming and requires significant expertise.

Generative language models like GPT-3 [5] on the surface appear very simple: they take a text-based input called a prompt, convert it to a list of tokens, predict what tokens are likely to follow the prompt, and convert those predicted tokens back into a text-based output. GPT-3 was trained on a large text corpus, and is capable produce novel outputs that appear almost human-like. For example, providing the previous source code with the prompt: *"# Explain the following javascript code line by line as a bulleted list:"*, GPT-3 can produce output such as the following (outputs are not deterministic, and the following was a typical output we observed):

- The first line imports the 'serve' function from the 'server' module in the 'std' library from version 0.140.0 of the Deno runtime.
- The second line declares a 'count' variable and initializes it to '0'.
- The third line defines a 'handleRequest' function that takes a 'request' parameter. This function increments the value of 'count' if the 'request' method is 'POST'.
- The fourth line returns a new 'Response' object containing the value of 'count'.
- The fifth line calls the 'serve' function, passing in the 'handleRequest' function and an options object. The options object sets the 'port' property to '7777'.

Similarly, by changing the prompt to: *"# Give a detailed explanation of the purpose of the following Javascript code:"*, we observed the following output:

> The purpose of this code is to create a simple server that counts how many POST requests it receives.

The ability to automatically generate such explanations appears to have great potential. To explore this in practice, we generated explanations for code listings in an online e-book for a web development course, and deployed these to students.

The novel contributions of our work are as follows:

(1) An in-class deployment and evaluation of LLM-generated code explanations
(2) An in-class comparison of multiple code explanation types

2 RELATED WORK

2.1 Pedagogical methods for explaining code

The ability to write and modify code, understand its purpose, and articulate its functionality are key skills that CS students must develop [7, 27, 43]. Explanations benefit students in multiple ways. Explanations help students to make connections and to develop their own understanding of how a code snippet executes [26]. Explanations can also reduce stress and effort with debugging one's own code, as well as improve learning [14]. Explanations also help students improve their reasoning when writing their own code [27].

Various pedagogical methods have been explored to help students develop their code comprehension skills. For example, the BRACElet project provided students with 'explain in plain English' type questions to encourage students to explain the purpose of code fragments at an abstract level [44]. These 'self-explanations' can provide short and long-term learning benefits for students [27, 42]. Code tracing is another popular approach to help students understand how their code executes and "to predict its behavior, the changes it makes to the computer's internal memory state (such as variables and data structures), and its output" [36]. In classrooms, collaborative activities can also facilitate peer explanations. For example in pair programming, students work together which requires them to often explain their code and their thinking process to their partner [16]. Similarly, misconception-based feedback techniques encourage peers to follow prompts based on common misconceptions to guide peer discussions about code [19].

These techniques strive to help students develop their understanding of code that they encounter when learning, and help them develop explanations of said code. Explaining code to peers and tracing the execution of a code snippet are cognitively demanding tasks, however, and not all students engage with such activities. To help, instructors can create explanations for students as examples, although providing personalized explanations to every student in the class is time consuming [40]. Therefore, automatic feedback generation of explanations may facilitate learning at scale.

2.2 Providing automated guidance to students

To provide personalized feedback and explanations to an entire class, intelligent tutoring systems (ITSs) [30] have become increasingly common in computer science classrooms. Initially, these systems focused on reducing the grading effort on instructional staff through the use of automated grading systems [1, 18, 31]. However, these grading systems have become increasingly focused on providing formative feedback such as hints [32, 35] error-specific feedback [25, 35] to help students avoid getting stuck in counterproductive behaviors when their answers are incorrect [4]. ITSs can also guide students through thinking processes. For example, PythonTutor traces the execution of code to help students understand how variables are assigned and modified as code executes [15]. To help students learn new syntax and programming patterns, Tutorons define terms in real-time using a heuristic approach [17]. These automated methods augment the code that students see to help them reason about code. Alternatively, Whyline encourages active engagement and critical thinking for students by generating reflection prompts [20].

Across these examples, ITSs help instructors reduce time and effort when it comes to grading and providing feedback [40]. By evaluating the correctness of students' code and giving them feedback to help them proceed, these ITS have the potential to free instructors time to work directly with students and to explain complex coding concepts in more detail.

2.3 Large language models enter the classroom

Large Language Models (LLMs) are neural networks that are trained on a massive corpus of human-generated text. LLMs have the capability to generate diverse and high-quality text-based responses to natural language prompts. This capability has inspired researchers to experiment with using them in classrooms to support both students and instructors [39]. Across multiple domains, LLMs have been used to support the writing process [13, 45], to generate code [2, 11, 37], to engage students in dialogue [39], and to explain concepts in plain English [24].

In computer science classrooms, LLMs are beginning to be used to generate assignments [37], help students to write their code [3, 41], and to generate explanations of code snippets to facilitate learning [24]. Previous research has demonstrated how LLMs such as Codex can generate code for students based on a specification [11] as well as generate assignments for instructors [37]. Explanations of code can also be generated by LLMs [24, 37]. When using Codex for generating code explanations, researchers found that line-by-line explanations could be generated, but they raised concerns about the explanation quality [37]. When using GPT-3, researchers showed that multiple explanation types could be generated [24]. However, the explanations in both studies were not yet evaluated by students nor deployed to classroom contexts. In this paper, we expand on previous research by formally evaluating explanations generated by an LLM in a classroom context.

3 STUDY CONTEXT

The study was conducted in-situ as a part of a web software development course offered by Aalto University. Aalto University is a research-oriented University in Finland, where the majority of the students are native Finnish speakers, although the students, in general, possess the capability to work and study in English. The web software development course is offered to the students in English, and it is taken during the second year of the Bachelor in Computer Science program. Students from other programs can also take the course.

The course teaches principles of web software development with a focus on server-side functionality such as designing and building APIs, working with databases, and creating pages using template-based languages rendered on the server. The workload of the course is 5 ECTS[1]. The course relies on JavaScript as the programming language and Deno as the runtime.

The course uses an online e-book written by the course instructor that contains interleaved theory and practice parts, interspersed with code examples, programming exercises, and quizzes. The grading in the course is based on completed coursework and two larger projects. Annually, approximately 300 students complete the course.

[1]European Credit Transfer System. 5 ECTS means approximately 125 to 150 hours of work, although the actual workload varies based on students' backgrounds and skills.

4 METHODOLOGY

4.1 Research questions

To explore the potential of automatically generated code explanations, we investigate the following research questions:

RQ 1: How do students engage with the generated explanations, and which types of explanations do they use most frequently? For what types of code snippets do students request explanations?

RQ 2: What are the characteristics of explanations that students rate as most and least helpful?

4.2 Prompt engineering and choosing an LLM

We created code explanations for a selection of code snippets of the web software development course using both GPT-3 [5] and Codex [6]. Our research team engineered multiple prompts for both LLMs to evaluate which generated better explanations. This prompt engineering process involved trial and error to determine which prompt phrases and level of detail resulted in the best responses from the LLM. We discovered that Codex tended to generate less helpful and more verbose explanations than GPT-3. Many of the explanations generated by Codex also included code which was unwanted, as our objective was to create natural language code explanations. GPT-3 on the other hand was more capable of consistently generating high-quality explanations, perhaps due to being more suitable for generating natural language output. We used the default model parameters for the GPT-3 Davinci model (i.e.: text-davinci-002). Examples of explanations for three programs generated by GPT-3 and Codex are shown in Figure 1.

Codex: # Summarize and explain the goal of the above code

(1) A web server is listening on port 7777. The handleRequest function is called when a request is accepted. What is the structure of the Deno module system? Deno supports ES Modules.

(2) The goal of this code is to create a server that will count the number of times that the server receives a POST request. It will then return the number of times that the server has received the POST request.

(3) The goal of the above code is to destructure the object obj. What is the value of first and second? first = "hello", second = "world!". What is the value of obj? obj = { first: "hello", second: "world!" }; What is the value of { first, second }? { first, second } = { first: "hello", second: "world!" };

GPT-3: # Summarize and explain this code snippet

(1) This code snippet creates a web server that listens on port 7777 and returns "Hello world!" for every request.

(2) This code snippet creates a server that listens for POST requests on port 7777. For each POST request received, the server increments a counter and returns the current value of the counter in the response.

(3) This code snippet is using object destructuring to assign the values of the properties in the object "obj" to the variables "first" and "second". It then logs the values of those variables to the console.

Figure 1: A comparison of the explanations generated by two large language models (GPT-3 and Codex) for three code snippets: (1) a 'hello world' server, (2) a server that counts POST requests, and (3) an example of object destructuring. The prompts used are shown. GPT-3 tended to produce more concise and consistently helpful explanations than Codex.

4.3 Augmenting the e-book with explanations

We generated explanations for 13 code snippets in the online e-book. For each code snippet, three types of explanations were generated (line-by-line explanation, summarization, listing concepts). As LLMs can create varying content, five code explanations were created for each of the three explanation types, for each of the 13 code snippets. This lead to a total of $13 * 5 * 3 = 195$ code explanations. The explanations were added to two chapters (Chapter 3 and Chapter 11) in the e-book so that students could optionally view them. For the present experiment, three buttons—each corresponding to one type of explanation—were added side by side below each code snippet with LLM-generated code explanations.

When pressing a button, students were shown the explanation corresponding to the type indicated by the button. When LLM-generated content was shown, students were also given a feedback form that asked them to rate the content and highlight any issues they observed in the explanation. The content also had a statement making it explicit to students that the content was automatically generated by an AI.

4.4 Measures

Our analysis focused on how students interacted with the explanations and their subject ratings of the quality and relevance of the generated explanations. We collected the explanations associated with each code snippet, and we collected behavior data which included logging a timestamp when students opened and closed an explanation for a given code snippet. This enabled our team to compute the **explanation view time.** We also collected the **number of views** that each explanation received. When viewing explanations, students were presented with a form that asked them to provide **subjective ratings** on one of the following statements along a likert scale: 1) *"the explanation matched the code"* and 2) *"I knew what the code did before viewing the explanation."* They were also asked to rate one of the following statements: 1) *"The explanation was useful for my learning."* and 2) *"The explanation was useful for me."*

5 RESULTS

5.1 Analysis of students' viewing behaviors

The study was run for three weeks during the summer of 2022. During the study, 176 explanations were viewed by 58 students of the 116 students who viewed the e-book. Of the students who viewed an explanation, they viewed 3.0 ($sd = 2.7$) explanations on average. Participants also spent on average 51.5 ($sd = 49.5$) seconds viewing each explanation with a maximum of 268 seconds.

5.1.1 For longer code snippets, students viewed explanations longer. We observed a strong positive correlation ($r(11) = 0.92, p < .05$) between the amount of time spent viewing an explanation and the length of the corresponding code snippet.

5.1.2 The explanations for the first code snippet were the most viewed, and more complex code snippets received more views. Of the 95 students who viewed Chapter 3, 42.1% of students viewed an explanation for the first code snippet. For the remaining 8 code snippets in the chapter students who visited the page were less likely to view an explanation. On the second page with a code snippet in the chapter, 9.4% of students viewed an explanation. On the last two pages of the chapter, students viewed explanations at a rate of 10.7% and 10.3% respectively. However, in Chapter 11—which contained more challenging code snippets—students were much more likely to view the explanations. Across three pages the view rates were 35.3%, 32.3%, and 39.3%.

5.1.3 Line-by-line explanations were viewed more than other explanation types. While some code snippets were more viewed by students than others, students also viewed some explanations more than others. Students viewed line-by-line explanations the most frequently at 103 times. Summary explanations were viewed 39 times and concept explanations were viewed 34 times. Line-by-line explanations made up 58.5% of the explanation types viewed, making it the most popular explanation type. We acknowledge that this might be due to the order of the buttons in the e-book, where the button for a line-by-line explanation was the leftmost.

5.2 Analysis of students' ratings of explanations

Students rated the explanations along a 5-point Likert scale. We used a continuous slider with a default value of 3. Therefore, it is not clear whether students who rated an explanation 3 for both likert scale responses intended to rate the explanation or not. As result, we removed all ratings that had the default ratings for both likert responses. We also excluded two ratings with a low view time where the students could not have read the code explanation. This left us with a total of 45 ratings for our analysis.

5.2.1 Explanations matched the code and were useful for learning. Students rated that the explanations match the code ($\mu = 4.5, sd = .78, n = 29$), tended to already know what the code did ($\mu = 4.6, sd = .63, n = 16$), rated the explanations somewhat useful for learning ($\mu = 3.8, sd = 1.0, n = 18$) and somewhat useful for them ($\mu = 3.9, sd = 1.2, n = 28$). However, explanations were deemed less useful when students already knew what the code does. We observed a weak negative correlation between the perceived usefulness of the explanation (-0.37 for learning, -0.44 for me) and already knowing what the code does.

5.2.2 Line-by-line explanations were rated as least useful for learning. Despite their popularity, line-by-line explanations tended to receive lower ratings from students. Summarized in Figure 2, line-by-line explanations were perceived by students as being less useful for learning than summary explanations and concept explanations. We performed Kruskal-Wallis test for statistically significant differences between explanation type based on perceived usefulness, but obtained non-significant results ($p > .05$). We note, however, that the assumption of independence was partially violated and that the sample sizes were small.

5.2.3 Qualitative analysis of low-quality explanations. To better understand why students rated some explanations as low quality, we conducted an analysis on all explanations that were rated either a 2 or 1 out of 5. In almost every case, the explanations were correct; however, we observed the following undesirable qualities in some explanations: 1) the explanation was overly detailed and focused on mundane aspects of the code, 2) the explanation was the wrong type (e.g.: a concept explanation that read more like a line-by-line explanation), 3) the explanation mixed code and explanatory text.

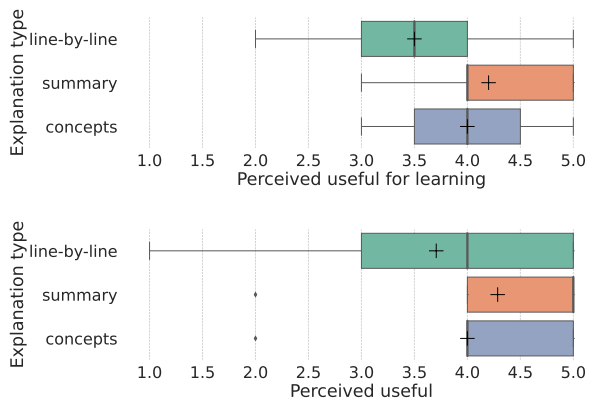

Figure 2: Boxplot of explanation usefulness ratings with + indicating mean. Although most viewed among students, line-by-line explanations were rated least helpful.

6 DISCUSSION

6.1 Generating code explanations

As part of our analysis, we compared and contrasted explanations generated by both GPT-3 and Codex. We observed that GPT-3 tended to create better quality explanations. The explanations from Codex tended to go off-topic and often included randomly generated code snippets. At times, Codex asked rhetorical questions. Overall, GPT-3 consistently generated explanations that were more useful and followed a standard structure. While few-shot learning is often recommended for LLMs [12], we observed that it was not particularly helpful for generating code explanations. The responses from both LLMs tended to overfit the structure of the response.

6.2 Engagement with code explanations

Around half of the students who opened the e-book engaged with explanations during our study. Some students engaged more with some code snippets and explanation types than with others. Curiosity and the complexity of the code snippets tended to drive student engagement. Students engaged most with the first code snippet explanation, likely to explore the new functionality and satiate their curiosity. Otherwise, the code snippets with the most views came later in the course when the code snippets were more complex. We also observed that students spent more time viewing explanations for longer code snippets.

These observations could be partially explained by the format and contents of the online e-book itself. The e-book is a coherent and self-contained entity, which has been in use for teaching web software development for several semesters prior to the introduction of the LLM-generated code explanations. It contains plenty of code samples and instructor-written code explanations, which guide students' work in course assignments. Our motives for including the code explanations into the e-book were to provide students with multiple different explanations of code, which in turn could help them in building a stronger understanding of the topic. Yet, it is clear that students did not utilize the code explanations to the extent that was expected by our team.

6.3 Code explanation usefulness and quality

Based on our preliminary findings, explanations appear to be helpful for learning. Overall, students rated the explanations as both relevant and useful for their learning. Students requested many more line-by-line explanations than any other type of explanation, although this could be in part due to the user interface. Regardless, line-by-line explanations were rated less helpful for learning by students in the study. Students seemed to prefer the summary explanations most. In future work, we plan to engage more deeply with other factors such as explanation length, clarity, and completeness. We suspect that more insight on how to improve the length, clarity, and completeness of explanations through methods such as prompt engineering may allow us to generate more useful explanations.

Knowing that LLMs have been found to produce incomplete code explanations also at a novice programming level [37], we briefly analyzed the generated code explanations to identify flaws. At a cursory glance, we did not observe any significant mistakes and the explanations were in general correct (although at times omitting details, as one would expect). This could be in part be related to the prompt engineering conducted as a part of this work; we iterated over prompts that have worked for us also in our prior experience with LLMs. We see collecting more subjective and qualitative data from the use of LLM generated code explanations as one way forward, as such data can help understand the variety of ways how code explanations can be understood and viewed.

6.4 Future directions

Based on our experiences, we envision a multitude of directions that could be investigated to increase the utility of LLM-generated code explanations in online e-books and in CS education in general.

6.4.1 Prompt engineering and personalization. In this study, we engaged in prompt engineering and evaluated prompts for two separate LLMs. Prompt engineering significantly contributes to the model performance [22], and hence prompt engineering should receive further attention when applying LLMs to produce code explanations. Recent research has suggested that LLMs can be used to generate a diverse body of code explanations [24], including the generation of analogies and fixing bugs, which would be meaningful to evaluate further in educational contexts. We also expect that personalizing the LLM output based on a student's prior experience or other preferences could benefit their learning. Ultimately, there are still many design decisions that need to be evaluated around generating the best prompts for learning. Ideally, we should show a student an explanation only when they need it, where the need could be evaluated e.g. through learner modeling, and the explanations could rely on topics that students find interesting.

6.4.2 Increasing engagement. Presently, the LLM-generated code explanations were available to students at the press of a button, but there were no incentives to engage with the code explanations. Based on prior research on e.g. code visualizations, which have highlighted the need for engagement with learning materials [28], we see that LLM-generated code explanations—and code explanations in general—would benefit from additional engagement beyond viewing (and potentially responding). As an example, as outlined in the engagement taxonomy [28], students could be asked questions

about the explanations, adapt the explanations, and create their own explanations. To provide further incentives, these could also be graded activities.

6.4.3 Learnersourcing and continuous improvement. Asking questions about code explanations, having students adapt the explanations, and having students create their own explanations could also be seen as a learnersourcing activity, which has the potential to lead to continuously improving learning resources. Learnersourcing has been used successfully in computing education to create multiple choice questions [8], programming exercises [9, 33], and SQL exercises [21]. We see learnersourcing as one way to improve explanation quality while encouraging students to more actively engage with explanations.

6.4.4 Live code explanations. A unique benefit of LLMs is that they create code explanations on demand. Online e-books already contain explanations in the text surrounding the examples which might explain why students found explanations matched the code, but only found them slightly useful. Explanations might best support students in contexts where explanations do not already exist. For example, when students write their own code, LLMs might provide feedback on their code and guide their debugging. In these contexts, students could learnersource additional help from their peers when the LLM-generated feedback is not sufficient. Such live code explanations could potentially increase engagement while adding to a database of code snippets that need feedback, LLM-generated explanations of such code, and student-generated explanations of such code. These could be used both to improve LLMs and inform educators about the common issues that students face, and what type of support helped students resolve these issues.

6.4.5 Browser extension. While the other future directions have mostly focused on using LLMs *within* e-books or in constrained learning environments, we see potential for LLMs to explain code in the wild. When confused, students search for help beyond the course materials and course staff, e.g. turning to forums such as StackOverflow [10, 34]. LLMs could generate code explanations for these external resources. For example, a browser extension could enable students to click on a code snippet on any web page to request an explanation of that code from an LLM. Similar browser extensions already exist to explain web visualizations to support informal learning [23]. If such functionality would be integrated into specific courses, and if students would give their permission, the data collected from the browser extension could also lead to insight into what sorts of external resources students use and seek to understand.

6.5 Limitations of work

Our preliminary results suggest that code explanations generated by LLMs matched the code well and were useful for learning. Students engaged with them, albeit less than we might have hoped. However, it is important to consider our results along with a number of limitations. First, this pilot study might be affected by selection bias: the students who chose to engage with the LLM-generated code explanations may not represent the "average" student in the course. For example, it is possible that they are more engaged than the average student; although, on the other hand, it is possible that struggling students are more likely to use this additional support. Relatedly, the interface presented the buttons in situ with the code snippets and it is not clear what effect if any the interface had on the discoverability of explanations and student engagement.

Related to the course context, although the course materials are in English, English is a second language for many students as the study was conducted in Finland. This may have affected how well students understand the explanations and their willingness to rate them. Furthermore, the e-book facilitates self-directed learning and therefore already contains explanations, which may have limited the need for more explanations.

For the data collection, the feedback form had a default value at the center of the Likert scale (3 out of 5), which was logged if the student closed the form. During data analysis, we noticed that there were some student feedback submissions that were created very quickly after opening the explanation, possibly signaling that the student did not intentionally rate the explanation using the form. As mentioned earlier, we removed 61 ratings where the default value was selected for both responses. It is possible that intentional ratings were removed in the process, potentially affecting the results.

Finally, our study is conducted in an online classroom context with a relatively small number of students. Considering these many limitations, our results are preliminary. They make a strong case for future research to be conducted in this area and highlight exciting new opportunities for future work.

7 CONCLUSION

In this work, we reported our experiences from using large language models (LLMs) to create code explanations and using them in a classroom setting. Our results suggest that while not all students utilized the created explanations, students tended to rate them as being useful for learning. We believe that these explanations might be even more helpful in settings where students do not already have a good understanding of the code which we discussed as future work. Ultimately, this work provides preliminary evidence that LLMs can be beneficial for students in CS classrooms. More work is needed to systematically investigate the design space of LLM-generated explanations in CS classrooms.

REFERENCES

[1] Kirsti M Ala-Mutka. 2005. A survey of automated assessment approaches for programming assignments. *Computer science education* 15, 2 (2005), 83–102.

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732 [cs.PL]

[3] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2022. Grounded Copilot: How Programmers Interact with Code-Generating Models. *arXiv preprint arXiv:2206.15000* (2022).

[4] Joseph E Beck and Yue Gong. 2013. Wheel-spinning: Students who fail to master a skill. In *International conf. on artificial intelligence in education*. Springer, 431–440.

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in neural information processing systems*. 1877–1901.

[6] 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).

[7] Kathryn Cunningham, Yike Qiao, Alex Feng, and Eleanor O'Rourke. 2022. Bringing "High-Level" Down to Earth: Gaining Clarity in Conversational Programmer Learning Goals. In *Proc. of the 53rd ACM Technical Symposium on Computer Science Education V. 1*. ACM, New York, NY, USA, 551–557.

[8] Paul Denny, John Hamer, Andrew Luxton-Reilly, and Helen Purchase. 2008. PeerWise: students sharing their multiple choice questions. In *Proceedings of the fourth international workshop on computing education research*. 51–58.

[9] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. Codewrite: supporting student-driven practice of java. In *Proceedings of the 42nd ACM technical symposium on Computer science education*. 471–476.

[10] Pierpaolo Dondio and Suha Shaheen. 2019. Is StackOverflow an Effective Complement to Gaining Practical Knowledge Compared to Traditional Computer Science Learning?. In *Proceedings of the 2019 11th International Conf. on Education Technology and Computers*. 132–138.

[11] James Finnie-Ansley, Paul Denny, Brett A Becker, Andrew Luxton-Reilly, and James Prather. 2022. The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In *Australasian Computing Education Conf.* 10–19.

[12] Tianyu Gao, Adam Fisch, and Danqi Chen. 2021. Making Pre-trained Language Models Better Few-shot Learners. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conf. on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, Online, 3816–3830.

[13] Katy Ilonka Gero, Vivian Liu, and Lydia Chilton. 2022. Sparks: Inspiration for Science Writing Using Language Models. In *Designing Interactive Systems Conf.* (Virtual Event, Australia) *(DIS '22)*. ACM, New York, NY, USA, 1002–1019.

[14] Jean M. Griffin. 2016. Learning by Taking Apart: Deconstructing Code by Reading, Tracing, and Debugging. In *Proc. of the 17th Annual Conf. on Information Technology Education*. ACM, New York, NY, USA, 148–153.

[15] Philip J Guo. 2013. Online python tutor: embeddable web-based program visualization for cs education. In *Proc. of the 44th ACM technical symposium on Computer science education*. 579–584.

[16] Brian Hanks, Sue Fitzgerald, Renée McCauley, Laurie Murphy, and Carol Zander. 2011. Pair programming in education: a literature review. *Computer Science Education* 21, 2 (2011), 135–173. https://doi.org/10.1080/08993408.2011.579808

[17] Andrew Head, Codanda Appachu, Marti A Hearst, and Björn Hartmann. 2015. Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 3–12.

[18] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of recent systems for automatic assessment of programming assignments. In *Proc. of the 10th Koli calling international conf. on computing education research*.

[19] Cazembe Kennedy, Aubrey Lawson, Yvon Feaster, and Eileen Kraemer. 2020. Misconception-Based Peer Feedback: A Pedagogical Technique for Reducing Misconceptions. In *Proceedings of the 2020 ACM Conf. on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE '20)*. ACM, New York, NY, USA, 166–172.

[20] Amy J Ko and Brad A Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 151–158.

[21] Juho Leinonen, Nea Pirttinen, and Arto Hellas. 2020. Crowdsourcing Content Creation for SQL Practice. In *Proceedings of the 2020 ACM Conf. on Innovation and Technology in Computer Science Education*. 349–355.

[22] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2021. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *arXiv preprint arXiv:2107.13586* (2021).

[23] Stephen MacNeil, Parth Patel, and Benjamin E Smolin. 2022. Expert Goggles: Detecting and Annotating Visualizations using a Machine Learning Classifier. In *The Adjunct Publication of the 35th Annual ACM Symposium on User Interface Software and Technology*. 1–3. https://doi.org/10.1145/3526114.3558627

[24] Stephen MacNeil, Andrew Tran, Dan Mogil, Seth Bernstein, Erin Ross, and Ziheng Huang. 2022. Generating Diverse Code Explanations Using the GPT-3 Large Language Model. In *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 2*. ACM, New York, NY, USA, 37–39. https://doi.org/10.1145/3501709.3544280

[25] Samiha Marwan, Ge Gao, Susan Fisk, Thomas W. Price, and Tiffany Barnes. 2020. Adaptive Immediate Feedback Can Improve Novice Programming Engagement and Intention to Persist in Computer Science. In *Proc. of the 2020 ACM Conf. on International Computing Education Research*. ACM, New York, NY, USA, 194–203.

[26] Samiha Marwan, Nicholas Lytle, Joseph Jay Williams, and Thomas Price. 2019. The Impact of Adding Textual Explanations to Next-Step Hints in a Novice Programming Environment. In *Proceedings of the 2019 ACM Conf. on Innovation and Technology in Computer Science Education*. ACM, New York, NY, USA, 520–526.

[27] Laurie Murphy, Sue Fitzgerald, Raymond Lister, and Renée McCauley. 2012. Ability to 'explain in Plain English' Linked to Proficiency in Computer-Based Programming. In *Proceedings of the Conf. on International Computing Education Research (ICER '12)*. 111–118. https://doi.org/10.1145/2361276.2361299

[28] Niko Myller, Roman Bednarik, Erkki Sutinen, and Mordechai Ben-Ari. 2009. Extending the engagement taxonomy: Software visualization and collaborative learning. *ACM Transactions on Computing Education (TOCE)* 9, 1 (2009), 1–27.

[29] Greg L Nelson, Benjamin Xie, and Amy J Ko. 2017. Comprehension first: evaluating a novel pedagogy and tutoring system for program tracing in CS1. In *Proc. of the 2017 ACM conf. on international computing education research*. 2–11.

[30] John C Nesbit, Olusola O Adesope, Qing Liu, and Wenting Ma. 2014. How effective are intelligent tutoring systems in computer science education?. In *2014 IEEE 14th international conference on advanced learning technologies*. IEEE, 99–103.

[31] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2022. Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Transactions on Computing Education (TOCE)* (2022).

[32] Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas Guibas. 2015. Autonomously Generating Hints by Inferring Problem Solving Policies. In *Proc. of the Second (2015) ACM Conf. on Learning @ Scale*. ACM, 195–204.

[33] Nea Pirttinen, Vilma Kangas, Irene Nikkarinen, Henrik Nygren, Juho Leinonen, and Arto Hellas. 2018. Crowdsourcing programming assignments with CrowdSorcerer. In *Proceedings of the 23rd Annual ACM Conf. on Innovation and Technology in Computer Science Education*. 326–331.

[34] Jaanus Pöial. 2020. Challenges of Teaching Programming in StackOverflow Era. In *International Conf. on Interactive Collaborative Learning*. Springer, 703–710.

[35] Thomas W Price, Yihuan Dong, and Dragan Lipovac. 2017. iSnap: towards intelligent tutoring in novice programming environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on computer science education*. 483–488.

[36] Ruixiang Qi and Davide Fossati. 2020. *Unlimited Trace Tutor: Learning Code Tracing With Automatically Generated Programs*. ACM, 427–433.

[37] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models. In *Proceedings of the 2022 ACM Conf. on International Computing Education Research - Volume 1*. ACM, New York, NY, USA, 27–43.

[38] Dominik Sobania, Martin Briesch, and Franz Rothlauf. 2021. Choose Your Programming Copilot: A Comparison of the Program Synthesis Performance of GitHub Copilot and Genetic Programming. https://doi.org/10.48550/ARXIV.2111.07875

[39] Anaïs Tack and Chris Piech. 2022. The AI Teacher Test: Measuring the Pedagogical Ability of Blender and GPT-3 in Educational Dialogues. *arXiv preprint arXiv:2205.07540* (2022).

[40] Zahid Ullah, Adidah Lajis, Mona Jamjoom, Abdulrahman Altalhi, Abdullah Al-Ghamdi, and Farrukh Saleem. 2018. The effect of automatic assessment on novice programming: Strengths and limitations of existing systems. *Computer Applications in Engineering Education* 26, 6 (2018), 2328–2341.

[41] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *CHI Conf. on Human Factors in Computing Systems Extended Abstracts*. 1–7.

[42] Arto Vihavainen, Craig S Miller, and Amber Settle. 2015. Benefits of self-explanation in introductory programming. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. 284–289.

[43] Wengran Wang, Yudong Rao, Rui Zhi, Samiha Marwan, Ge Gao, and Thomas W. Price. 2020. Step Tutor: Supporting Students through Step-by-Step Example-Based Feedback. In *Proceedings of the 2020 ACM Conf. on Innovation and Technology in Computer Science Education*. ACM, New York, NY, USA, 391–397.

[44] Jacqueline L. Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P. K. Ajith Kumar, and Christine Prasad. 2006. An Australasian Study of Reading and Comprehension Skills in Novice Programmers, Using the Bloom and SOLO Taxonomies. In *Proc. of the 8th Australasian Conf. on Computing Education - Volume 52*. Australian Computer Society, Inc., AUS, 243–252.

[45] Ann Yuan, Andy Coenen, Emily Reif, and Daphne Ippolito. 2022. Wordcraft: Story Writing With Large Language Models. In *27th International Conf. on Intelligent User Interfaces*. 841–852.