Evaluating LLM-generated Worked Examples in an Introductory Programming Course

Breanna Jury
University of Auckland
Auckland, New Zealand
bjur781@aucklanduni.ac.nz

Angela Lorusso
University of Auckland
Auckland, New Zealand
alor903@aucklanduni.ac.nz

Juho Leinonen
University of Auckland
Auckland, New Zealand
juho.leinonen@auckland.ac.nz

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

Andrew Luxton-Reilly
University of Auckland
Auckland, New Zealand
andrew@cs.auckland.ac.nz

ABSTRACT

Worked examples, which illustrate the process for solving a problem step-by-step, are a well-established pedagogical technique that has been widely studied in computing classrooms. However, creating high-quality worked examples is very time-intensive for educators, and thus learners tend not to have access to a broad range of such examples. The recent emergence of powerful large language models (LLMs), which appear capable of generating high-quality human-like content, may offer a solution. Separate strands of recent work have shown that LLMs can accurately generate code suitable for a novice audience, and that they can generate high-quality explanations of code. Therefore, LLMs may be well suited to creating a broad range of worked examples, overcoming the bottleneck of manual effort that is currently required. In this work, we present a novel tool, 'WorkedGen', which uses an LLM to generate interactive worked examples. We evaluate this tool with both an expert assessment of the content, and a user study involving students in a first-year Python programming course (n = ~400). We find that prompt chaining and one-shot learning are useful strategies for optimising the output of an LLM when producing worked examples. Our expert analysis suggests that LLMs generate clear explanations, and our classroom deployment revealed that students find the LLM-generated worked examples useful for their learning. We propose several avenues for future work, including investigating WorkedGen's value in a range of programming languages, and with more complex questions suitable for more advanced courses.

CCS CONCEPTS

• **Software and its engineering** → **Programming by example**; • **Social and professional topics** → **Software engineering education**.

KEYWORDS

LLM, large language models, chat-GPT, GPT-3.5, computing education, worked examples, CS1

ACM Reference Format:
Breanna Jury, Angela Lorusso, Juho Leinonen, Paul Denny, and Andrew Luxton-Reilly. 2024. Evaluating LLM-generated Worked Examples in an Introductory Programming Course. In *Australian Computing Education Conference (ACE 2024), January 29–February 02, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3636243.3636252

1 INTRODUCTION

Worked examples are a well-established method of improving student learning by reducing cognitive load [13, 15, 23, 31]. In brief, a worked example is a resource given to students where a course-related question is answered, and the individual steps in the problem-solving process are illustrated step-by-step. Learning via worked examples helps students to build new connections and aids in the storage and retrieval of key concepts [32]. Worked examples are time and effort-intensive to create, and therefore instructors tend to only create a few examples for students to learn from.

Large Language Models (LLMs) are a form of Artificial Intelligence (AI) and result from training neural networks on enormous quantities of textual data [4]. LLMs are able to rapidly generate novel human-like outputs, including source code. This capability has sparked great interest in the computing education community [6, 9, 27], and popular public-access LLMs such as OpenAI's ChatGPT have raised the profile of LLMs to a general audience.

There is a wealth of literature on worked examples and their efficacy in the specific domain of programming [34, 37]. Due to the relatively recent emergence of LLMs, although they are beginning to be evaluated and integrated into a range of learning tools [16, 20, 33], to our knowledge, they have not been applied to the problem of automatically generating worked examples. Given the documented capabilities of LLMs for generating code [10] and associated code explanations [18], we see strong potential for LLMs to be used for this purpose. The generation and evaluation of a wide range of worked examples, on demand, has the potential to greatly improve student learning.

This research progresses in two stages. Firstly, to identify effective approaches for generating high-quality worked examples, we evaluate the quality of worked examples generated via a range of prompting strategies. This first stage of the research is guided by the following two research questions:

- **RQ1:** How well can LLMs generate clear explanations in a specific knowledge domain?
- **RQ2:** How effectively can LLMs decompose a worked example into well-defined steps?

Secondly, using our best generation strategy, we deploy LLM-generated worked examples to students in an introductory programming course using a bespoke tool. Our web-based application, named 'WorkedGen', provides a front-end user interface to generate and view worked examples. WorkedGen allows the user to enter a programming language and a question for which a worked example is generated. Users can then view the example step-by-step, and interact with the LLM by asking questions or clicking on keywords and lines of code. We use WorkedGen to further evaluate the usefulness of LLM-generated worked examples with the following two research questions:

- **RQ3:** How do novices perceive the usefulness of on-demand LLM-generated worked examples?
- **RQ4:** How do novices interact with worked examples, and how do they perceive the usefulness of these interactions?

2 RELATED WORK

2.1 Worked Examples

Worked examples are an effective way for learners to understand complex ideas. They are shown to decrease learners' cognitive load and therefore aid in the retrieval and storage of concepts [32]. Students benefit from an opportunity to view concepts before having to put them into practice [12].

2.1.1 Worked Examples and Cognitive Load. Cognitive load theory defines three types of load: intrinsic, which relates to the difficulty of content; extraneous, the processing of unproductive information; and germane, the load imposed by the learning process [13]. If a person's cognitive load is exceeded, their learning will be hindered [15, 23]. Kalyuga et al. [15] showed that using worked examples effectively increases student learning by reducing the extraneous load and optimizing the germane load.

2.1.2 Ways of Optimizing Worked Examples. Several ways of optimizing worked examples have been proposed. Schwonke et al. [30] investigated worked examples enriched with fading. This is a method where parts of the example are hidden and only become revealed as the learner progresses. They determined that students required less time to complete problem-solving activities after being exposed to these worked examples, and in some cases developed a deeper conceptual understanding. Margulieux et al. [22] used a similar method of creating questions with subgoals and again found that learners performed better than with non-optimized worked examples. Muldner et al. [24] supported the use of fading and also speculated on the helpfulness of forced self-explanation by learners. Worked examples are a time-intensive resource to produce, and as such, learners tend to have access to only a small number of worked examples. Some work has been done on the concept of isomorphic questions to reduce the effort needed to produce worked examples [25, 38]. This shows promise but still requires significant manual effort.

2.1.3 Constructing Worked Examples. Atkinson et al. [2] proposed that worked examples should be constructed using the following steps: they should be divided into three to ten subsections in order to reduce the extraneous cognitive load; several examples should be used with increasing complexity to increase germane learning; and, contexts should vary so that the learner does not falsely link the context with the concept.

2.1.4 Worked Examples and Programming. In the context of programming education, the beneficial effects of worked examples align with the effects reported in other disciplines [31]. Both Zhi et al. [37] and Vieira et al. [34] demonstrate, through the use of A/B testing with an experimental and control group, how worked examples increase learning, especially in novice programmers.

2.2 Large Language Models

Large Language Models are a category of Artificial Intelligence in the field of Natural Language Processing (NLP). In recent years, LLMs have revolutionized the field by achieving high performance across a range of language tasks. These models are trained on large amounts of data and can generate high-quality human-like output. In this section, we briefly examine the performance of LLMs in generating both natural language output and source code and describe the prompting patterns that have been developed to optimize LLM performance. Although there are myriad opportunities offered by the application LLMs, researchers have also called for increased focus on the ethics of their usage, particularly when used in educational settings [3, 27].

2.2.1 Generating Natural Language. MacNeil et al. [21] investigated GPT-3 and found that it can generate code explanations at multiple difficulty levels. Leinonen et al. [19] found that error messages generated by Codex can be helpful, although they are inaccurate in some cases and should not be taken as a source of truth. More recently, MacNeil et al. [20] generated three types of explanations with GPT-3 and Codex: line-by-line explanations; lists of important concepts; and high-level summaries. They summarized two key findings: GPT-3 is more effective at generating code explanations than Codex; and, students found the high-level summaries most helpful. They also raised the possibility that future work could investigate personalizing the outputs to each student. Korinik [17] found that GPT-3 was highly useful in brainstorming, synthesizing text, and extracting data from text. This was measured by the quality of the generated text, the relevance, and the amount of real information it produced. They reported that GPT-3 frequently hallucinated when asked to produce reviews of literature, and it required large amounts of human oversight in such tasks.

2.2.2 Generating Code. Codex by OpenAI was one of the first popular open-access LLMs for code generation. Chen et al. [5] found that Codex created successful functions for 28.8% of problems, and 72% when repeated sampling was used. Ross et al. [28] concluded that Codex was accurate (i.e., that it generated a valid solution) in 80% of cases. However, Vaithilingam et al. [33] noted that care must be taken to validate all generated code. In computing education, Finnie-Ansley et al. [10, 11] found that the performance of Codex fell within the top quartile of students in both CS1 and CS2 programming courses when asked to solve exam questions.

In a study conducted by Kazemitabaar et al. [16], students given access to Codex performed better than students without access and reported less stress in the learning environment. Denny et al. [7], Jonsson and Tholander [14], and Sarsa et al. [29] concluded that the accuracy and usefulness of code generated was dependent on the prompts given to the LLM, so attention to prompting strategy is valuable.

2.2.3 Optimization of Prompts. One technique used to improve LLM outputs is few-shot learning. This is the process of giving the LLM examples of inputs and corresponding desired outputs to guide it toward similar generation behaviour. Ahmed and Devanbu [1] found that Codex trained on ten few-shot examples outperformed all fine-tuned models. Poesia et al. [26] also noted the positive effects of few-shot training. Wu et al. [36] proposed a novel method of prompting called chaining, through which prompts are fed step-by-step, with each output being used as part of the next input. This method saw better outcomes than one-step prompts 82% of the time. White et al. [35] catalogued several methods of prompt engineering which ask the LLM to produce output in a given pattern. Some notable patterns are the *Persona* pattern, where the LLM acts as a type of person or role, *Question Refinement*, where the LLM will generate better versions of the input, and *Context Manager*, where the LLM is instructed to ignore certain scopes. These patterns create a framework to leverage the power of an LLM. To teach students how to use LLM prompts more effectively for code generation, Denny et al. [8] created a novel tool 'Promptly' which was well-received by students.

2.3 Gaps and Opportunities

Despite the recent interest in LLMs in computing education, there is currently no work directly exploring the use of LLMs for generating and deploying on-demand worked examples to students learning programming. Furthermore, there is a scarcity of user studies, particularly at a large scale, that evaluate student interactions with LLM-generated content.

In this work, we explore a novel system that uses an LLM to generate worked examples for students in early programming courses. Leveraging the power of an LLM allows for the creation of an almost unlimited number of examples without manual effort. Access to a large and varied repository of worked examples may be beneficial to students.

3 METHODS

3.1 Design

We designed WorkedGen as a tool for the user-friendly generation of and interaction with LLM-generated worked examples. Informed by the literature on worked example design, the tool was built to display each worked example in discrete steps (three to ten) [2]. For the purposes of our user study, a set of pre-defined questions relevant to the course context was included as a 'question bank' to allow participants to generate a worked example on-demand simply by selecting an existing question. In addition, users had the option to input their own questions and programming language, allowing for the generation of worked examples on arbitrary topics. Figure 1 shows these features on the WorkedGen home screen.

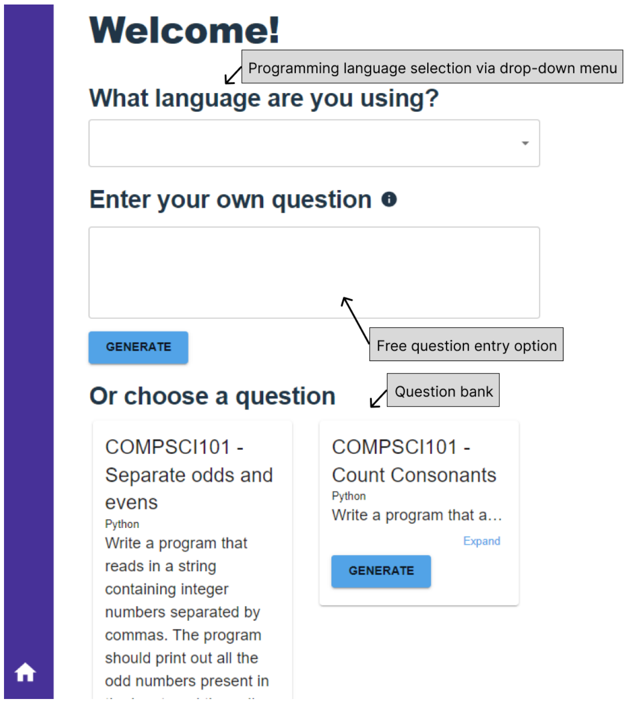

Welcome!
Programming language selection via drop-down menu
What language are you using?
Enter your own question
GENERATE
Free question entry option
Or choose a question
Question bank
COMPSCI101 - Separate odds and evens
Python
Write a program that reads in a string containing integer numbers separated by commas. The program should print out all the odd numbers present in
COMPSCI101 - Count Consonants
Python
Write a program that a...
Expand
GENERATE

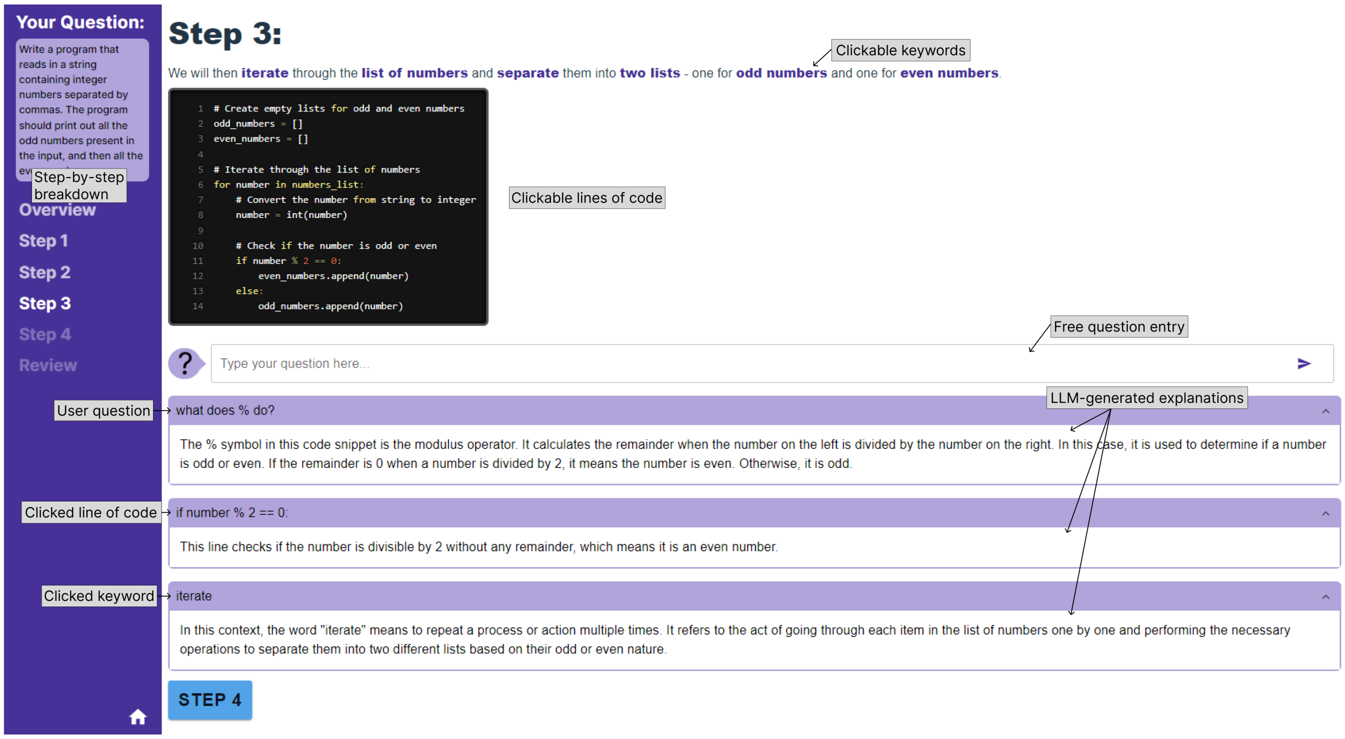
Figure 1: WorkedGen home screen, which contains the user input choices and provided question bank

The selected question would then be passed to the LLM to generate a worked example. Each step would be displayed on a separate page, following the principle of fading. The first step would be a general overview, with no specific steps or code shown, to prompt the user to think more deeply about the question. Each step would have code and accompanying explanations, as shown in Figure 2. Finally, the last step would be a review of the full code and an overarching explanation of the question. The exact prompting method is detailed in the next section.

WorkedGen was designed to allow for user interaction. This leverages the power of an LLM over traditional static worked examples, as the user can receive additional explanations as they need. Three kinds of LLM interaction exist in each worked example: keywords, lines of code, and questions. In each explanation, the LLM would be prompted to generate keywords for the text. These keywords would then be shown in bold and would be clickable. Each line of code can be clicked, and there is a free input box where the user can ask questions. All of these interactions are passed to the LLM along with the original context, and a short explanation is generated and displayed to the user. This is shown in Figure 2.

3.2 Prompt Development

3.2.1 Model Selection. Several general-purpose LLMs have been developed and made available for public use in recent years. Codex (OpenAI) was released in 2021 and was specifically designed for code generation [5]. It is less capable than other more recent models for generating natural language and thus is less suitable for generating worked examples. Access to Codex was also deprecated



Write a program that reads in a string containing integer numbers separated by commas. The program should print out all the odd numbers present in the input, and then all the ev

Step 3:

We will then **iterate** through the **list of numbers** and **separate** them into **two lists** - one for **odd numbers** and one for **even numbers**.

```python
# Create empty lists for odd and even numbers
odd_numbers = []
even_numbers = []

# Iterate through the list of numbers
for number in numbers_list:
    # Convert the number from string to integer
    number = int(number)

    # Check if the number is odd or even
    if number % 2 == 0:
        even_numbers.append(number)
    else:
        odd_numbers.append(number)
```

Type your question here...

what does % do?

The % symbol in this code snippet is the modulus operator. It calculates the remainder when the number on the left is divided by the number on the right. In this case, it is used to determine if a number is odd or even. If the remainder is 0 when a number is divided by 2, it means the number is even. Otherwise, it is odd.

if number % 2 == 0:

This line checks if the number is divisible by 2 without any remainder, which means it is an even number.

iterate

In this context, the word "iterate" means to repeat a process or action multiple times. It refers to the act of going through each item in the list of numbers one by one and performing the necessary operations to separate them into two different lists based on their odd or even nature.

Figure 2: WorkedGen step screen with additional explanations from user interaction with WorkedGen - questions, code lines and keywords

by OpenAI in early 2023, although some limited access has been retained for research purposes. Bard[1] (Google) is proficient at generating both code and natural language, and was released in response to OpenAI's release of ChatGPT in late 2022. Hugging Face[2] is a collection of free and open-source models, however, these are generally less powerful compared to the larger closed models, and require compute resources to run. GPT-3.5 Turbo (OpenAI) is a model that generates both code and natural language. It is relatively inexpensive, widely used, and one of the most well-studied and documented LLMs [4]. GPT-4[3] is an upgraded version of GPT-3.5, however, it is far more expensive (by a factor of roughly 20 at the time of writing). For the development and deployment of WorkedGen, we utilise the GPT-3.5 model from OpenAI. It has a well-documented API, and initial testing generated promising results for its usage in generating worked examples.

3.2.2 Prompts. WorkedGen has five points of interaction with the LLM. The first and most important point is the worked example generation. Sections of this worked example would be passed back into the LLM for the purpose of generating keywords. Three additional prompts were used to facilitate the extra explanations available when a user clicked a keyword or a line of code, or asked a question. Table 1 shows these five prompts.

[1]https://bard.google.com/
[2]https://huggingface.co/
[3]https://openai.com/gpt-4

We found that chaining the worked example and keyword prompts greatly improved the quality of the keywords. Attempting to generate both in the same prompt led to inconsistent outputs, but passing sections of the worked example back into the LLM as context along with the prompt shown in Table 1 generated consistently useful keywords. We used chaining again in the three user interaction prompts, which are used when the user clicks on keywords or lines of code or asks a question.

The generation of the worked example was the most complex prompt and was extremely important to the overall quality of WorkedGen. This prompt was developed over several iterations in order to improve the consistency of the output and the quality of the worked example. Two experts, with more than three years of experience in a range of languages including Python and Java and some experience in teaching, independently evaluated fourteen generated worked examples using the rubric shown in Table 2. This rubric was designed using key features of previous worked example research, including Atkinson et al. [2] and Skudder and Luxton-Reilly [31]. Any disagreements in the evaluation were discussed and resolved. Keywords such as 'worked example', 'novice' and 'written explanation and code' were used to direct the LLM to generate a high-quality worked example. Semantic features of the prompt such as 'START_OF_CODE', were used to generate consistent output to be processed for visualisation in WorkedGen.

The second part of the prompt involves one-shot learning, signposted with 'here is an example output to mimic'. Initially, a zero-shot approach was used, but this led to inconsistencies in the way

Function	Prompt
Worked Example	Write a worked example for the given question in {language}. Mark the start of any code with 'START_OF_CODE'. Mark the end of any code with 'END_OF_CODE'. Clearly mark each step with the heading 'STEP X'. Each step must contain both written explanation and code, and be separable from the other content. There should be minimum 3 and maximum 10 steps. Also include an overview at the beginning, with no code. Do not include steps in the overview. Include a review at the end with the complete code. The output should be suitable for a novice programmer. Here is an example output to mimic: Overview: Overview Explanation STEP 1: Step 1 Explanation START_OF_CODE Step 1 Code END_OF_CODE STEP 2: Step 2 Explanation START_OF_CODE Step 2 Code END_OF_CODE STEP 3: Step 3 Explanation START_OF_CODE Step 3 Code END_OF_CODE Review: Review Explanation START_OF_CODE Review Code END_OF_CODE The question is: ### {question} ###
Keywords	Give me keywords for this paragraph: {paragraph}
Keyword explanation	Given this paragraph: {paragraph}. Explain this word in context: {keyword}.
Code explanation	Given this code snippet: {snippet}. What does this line do: {codeLine}.
User question	Given this paragraph: {paragraph}. And this code snippet: {snippet}. Answer this question: {userQuestion}.

Table 1: Prompts used in WorkedGen — the purpose of the prompt and the text used as input to the LLM

Metric	Description
Code and explanations?	Does each step contain both code and a written explanation, and are these well connected?
Clear explanations?	Is each part of the code explained, and does the worked example flow in a clear and linear manner?
Code compiles?	Is the code able to compile and run?
Correct code?	Does the code generate the correct output?
Readable?	Does the code make sense, and does it use meaningful variable names?
Well commented code?	Are there valuable and meaningful comments in the code?
Meaningful steps?	Is each step an appropriate length, and tackles a discrete portion of the question? Are there between 3-10 steps?
Understandable?	Is the explanation understandable for a novice e.g. does not rely on prior knowledge?

Table 2: Rubric to evaluate worked examples - each metric and what is used to quantify it as met

the worked example was structured, which made it difficult to parse and display in WorkedGen. A one-shot learning approach was assessed in two different forms: using general and specific examples. The final prompt uses a general example, where the LLM is provided with the format of the output only. When using a specific worked example as a one-shot prompt, the LLM would often perform poorly and would not break the worked example down into distinct steps. Although we also explored few-shot learning, in which several examples are included in the prompt, this did not tend to yield better results and thus was not used due to additional cost and complexity. The final version of the worked example prompt, as shown in Table 1, generated consistent and high-quality output. Table 3 shows the output generated from the worked example prompt on a beginner Python question. Note that the review section has been excluded from the table, as it simply displays the full code shown across all the steps.

Question	Write a program that reads in a string containing integer numbers separated by commas. The program should print out all the odd numbers present in the input, and then all the even numbers.
Overview	This program will read in a string containing integer numbers separated by commas. It will then print out all the odd numbers present in the input, followed by all the even numbers.
Step 1	We will start by asking the user to input a string of numbers separated by commas. `# Ask the user to input a string of numbers` `# separated by commas` `numbers_string = input(` `"Enter a string of numbers separated by commas: ")`
Step 2	Next, we will split the input string into a list of individual numbers. `# Split the input string into a list of` `# individual numbers` `numbers_list = numbers_string.split(",")`
Step 3	We will then iterate through the list of numbers and separate them into two lists - one for odd numbers and one for even numbers. `# Create empty lists for odd and even numbers` `odd_numbers = []` `even_numbers = []` `# Iterate through the list of numbers` `for number in numbers_list:` `    # Convert the number from string to integer` `    number = int(number)` `    # Check if the number is odd or even` `    if number % 2 == 0:` `        even_numbers.append(number)` `    else:` `        odd_numbers.append(number)`
Step 4	Finally, we will print out the odd numbers followed by the even numbers. `# Print out the odd numbers` `print("Odd numbers:", odd_numbers)` `# Print out the even numbers` `print("Even numbers:", even_numbers)`

Table 3: Worked Example generated by WorkedGen and the question input - shows the explanation and code for each step if applicable

3.3 User Study

The evaluation of WorkedGen is guided by research questions 3 and 4, with the goal of understanding students' perspectives on the tool and the LLM-generated worked examples. The study was conducted in August 2023, with users from a large (~400 students) first-year

1. This tool was useful and helpful
2. I would use this tool again
3. This tool was error-free and easy to use
4. The explanations were clear and easy to understand
5. The problem was divided into logical steps
6. I found it helpful to see code and explanations in the same step
7. The code was functionally correct and readable
8. I found the ability to go back to previous steps helpful
9. I found the ability to click on keywords helpful
10. I found the ability to click on lines of code helpful
11. I found the ability to ask questions helpful

Table 4: List of survey questions

computer science course. Students were asked to use WorkedGen as part of their compulsory laboratory activities. Students were given a link to WorkedGen with instructions to test a question from the question bank, and to generate their own question. We received 337 pieces of written feedback, and the tool was used 787 times. This indicates that some students used the tool two to three times and that some students who used the tool did not submit written feedback.

Data from users was gathered in three ways. The first was through their interactions with WorkedGen. The questions they asked, the language they wanted outputted, and the number of steps the LLM generated for the worked example were recorded. There were three key ways that the user could interact with the tool: by clicking on a keyword, clicking on a line of code, or asking a question. The frequency and content of these clicks were recorded. At the end of each question, there was an optional survey that students could complete voluntarily. The survey consisted of 11 Likert scale questions to record the users' perspectives of the tool, as shown in Table 4. All questions were phrased so that 'strongly agree' would indicate a positive experience with WorkedGen.

Finally, a text entry section was included with the prompt 'In the text area below, please comment on the explanations with reference to what you have learned in the course so far (i.e., are the explanations helpful for revising or consolidating your knowledge)'. This allowed for qualitative data to be gathered on the users' experience.

4 RESULTS

4.1 How well can LLMs generate clear explanations in a specific knowledge domain?

Fourteen worked examples were generated using the final prompt in a range of languages and evaluated in various categories. Figure 3 shows an overview of the evaluation. The categories were chosen based on a review of important features of programming worked examples, such as functional correctness and understandable explanations. Additionally, the explanations should be suitable for novices as the target users of this study. The full rubric used can be seen in Table 2.

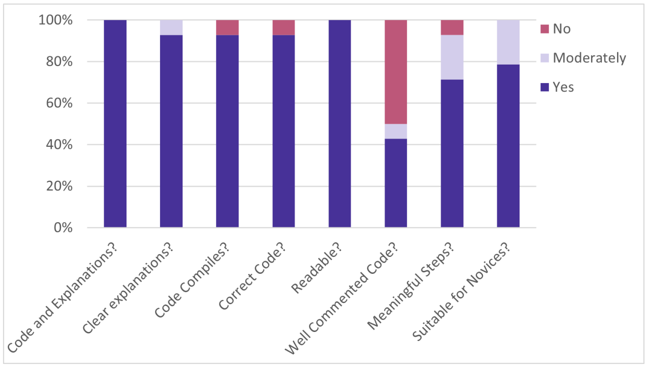


Figure 3: Expert evaluation of LLM-generated worked examples assessed against each metric

A common weakness of the worked examples was the lack of well-commented code. However, all code was readable with meaningful variable names. Only one worked example had incorrect code. This example also did not have clear explanations. The question submitted by a student was 'Write a script in bash that renames all files with a specific extension in a given directory. For example, change all ".txt" files to have a ".bak" extension. The key weakness was that it did not give any context or instruction on inputting the parameters, so the code was very difficult to use. This might be particularly harmful for novice users who do not have a strong understanding of the Bash language. Another common weakness of the explanations was their tendency to assume knowledge that novices may not have, such as inheritance in object-oriented programming.

4.2 How effectively can LLMs decompose a worked example into well-defined steps?

From an analysis of past work, it was determined that worked examples tend to have between three to ten steps, depending on the complexity of the question [2]. This metric was included in the prompt to generate worked examples using an LLM. Considering both the worked examples used in the expert evaluation and the user study, it was determined that most worked examples were generated with four steps, as seen in Figure 4. All but one example was generated with a number of steps in the ideal range.

The length and complexity of these steps varied depending on the question. Some steps were so short as to be almost meaningless, such as merely adding an import statement to the code. In a few occurrences, almost the entirety of the code was written in a single step. However, both of these issues were uncommon, and even when they occurred, the worked example was still readable and useful. The expert evaluation determined that 71% of the generated examples had a meaningful step decomposition. Additionally, 87% of novice users reported that the worked example they used was divided into logical steps.

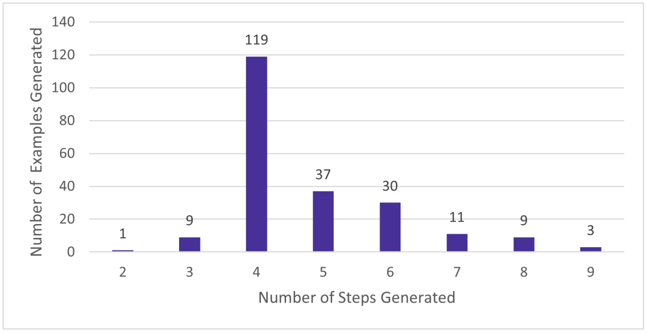


Figure 4: Frequency of the number of steps in worked examples generated by WorkedGen

4.3 How do novices perceive the usefulness of on-demand LLM-generated worked examples?

The user study recorded and analysed novices' attitudes towards the worked examples. This data came from questions 1-8 of the survey as shown in Table 4 and from the open-response questions. Figure 5 shows the survey responses to these questions. Overall, the response was positive. There was one outlier in the data who answered 'strongly disagree' to all survey questions, however, they did not include written feedback, so it is not possible to understand the reason for their negative experience. Students responded positively to having code and explanations shown at the same step and found the worked examples easy to understand.

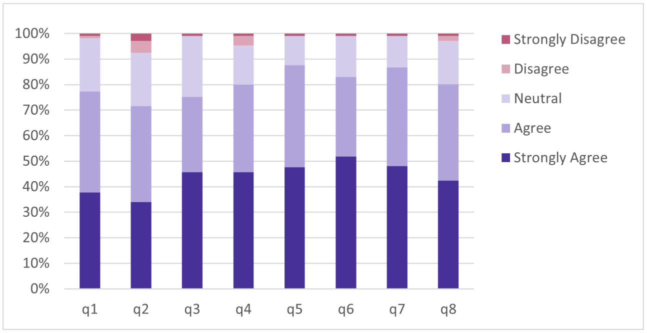


Figure 5: User results for questions 1-8 of our survey (survey questions as shown in Table 4)

The written feedback by students was also generally positive. All feedback was categorised into positive, neutral, or negative sentiment related to the experience overall, the step-by-step presentation, the code explanations, or the user interface (see Figure 6).

Only one user did not like the step-by-step nature of WorkedGen, *"...I do not think that the layout of the site is designed best for revision, as all information is not presented at once."* Other users found it very helpful for understanding complex questions: *"If I was asked to complete this question I would have been very confused where to begin, but after seeing each step, it clearly explains to me what needs to be done and in what order it should be in, especially the for...in loop."* One user noted how the steps being broken down helped to prompt self-explanation, *"...the way they were structured gave you the opportunity to figure out the next step before it came up so you could almost come up with the steps at the same time as you flick through the steps and double check that you got it right."*

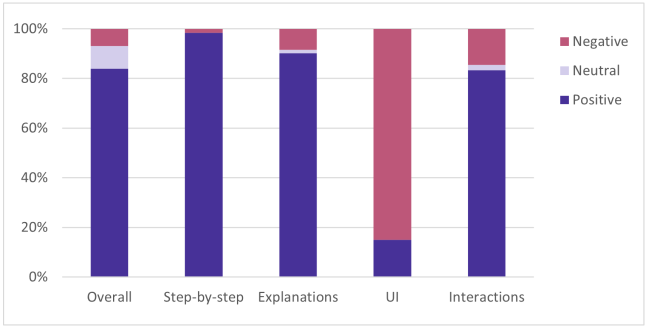


Figure 6: Sentiment analysis of written user feedback related to WorkedGen

The explanations were generally well received, with comments such as *"I think the explanation of 'WorkedGen' is very good, it is effective and concise."* However, some users wanted more background on the question and to explain the rationale of certain coding choices more in-depth, *"The explanations are useful for explaining what each step of the program does (at the least, providing an understanding of how the code functions). However, it doesn't help explain how to approach the problem-solving to DERIVE the steps initially. I.e. How do we break down this problem? What information have we been given? What tools do we have at our disposal? How can we apply them?"* This sentiment was echoed with some feedback on the code, with the LLM using concepts that they had not learnt in class. *"Although I've constantly seen a trend where AI seems to favour using def functions to solve problems. Whereas often we're solving problems where we can't use them since we haven't learned them yet."* Users found the code comments especially helpful in their understanding, with one user stating that they *"...found the explanations easy to understand in the two codes that were provided in the WorkedGen, especially because there were comments that explained what job each section of the code does."*

A feature that received a lot of comments but is not strictly related to the research of LLM-generated worked examples is the user interface of WorkedGen. Some users found it unintuitive to use, *"It took me a while to realise that each line of the code is clickable and is accompanied by explanations — this feature has the potential to be very useful, but wasn't immediately obvious."* WorkedGen wasn't suitable for mobile devices and sometimes conflicted with a user's colour settings, *"The dark mode in Safari was a bit hard to read as black text was used."* However, some users commented on positive aspects of the user interface, saying that *"The UI is interactive and engaging"*, and that it helped their understanding of the examples: *"I also quite like the colors of text and how key words are bold and easy to read."*

Most students used the provided questions to use WorkedGen, however, there were 192 uses of the custom generation tool. Some questions were conventional questions used for a worked example, such as *"Write a Python function that takes a list as input and returns the average of all even numbers in the list."* However, some questions were more general, such as *"I don't know when it is preferred using for loop or while loop"*. This was an interesting use case, and while it was not what WorkedGen was designed for, the LLM still handled it well by giving an example of where each would be preferred. Users of this feature generally rated the tool highly, as seen in Figure 7.

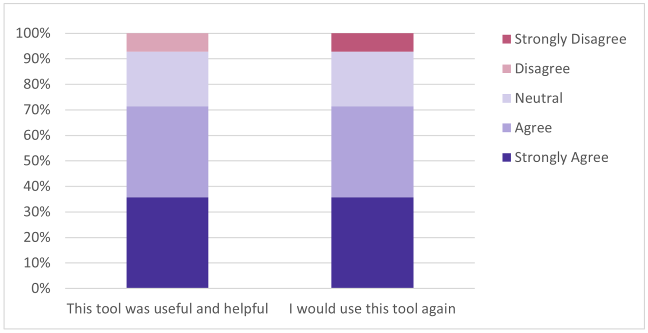


Figure 7: Two survey question responses of users who used the custom question generation tool

4.4 How do users interact with worked examples, and how do they perceive the usefulness of these interactions?

The interaction of users was recorded by tracking the clicks that they made on WorkedGen, and through questions in the survey and open-response questions. There were three key points of interaction: clicking on keywords, clicking on lines of code (LOC), or asking questions throughout the worked example. Table 5 shows the occurrences of clicks. Of the 787 uses of WorkedGen, only 201 uses recorded any clicks. This indicates that the click feature may not have been intuitive, or that many users did not require additional explanations. If the no-click uses are omitted, then there was an average of two clicks per use.

	Keywords	LOC	Questions	Total
Count	77	176	165	418
Mean	0.10	0.22	0.21	0.53
Modified Mean	0.38	0.88	0.82	2.08

Table 5: Click data from user interactions with WorkedGen — Modified mean excludes users who did not interact with WorkedGen

The survey results indicate that users found this interaction helpful, with the ability to ask questions rated the most positively. In general, a user's overall rating of the tool rose alongside the number of times they interacted with the tool, as shown in Figure 8. Negative responses were often due to the user interface of WorkedGen, with one user commenting *"I didn't realise I could click on keywords, click on lines of code, and ... didn't know what the questions were for."*

Users found it helpful to receive more explanations on sections that they found tricky. *"I think the explanations were wonderful*

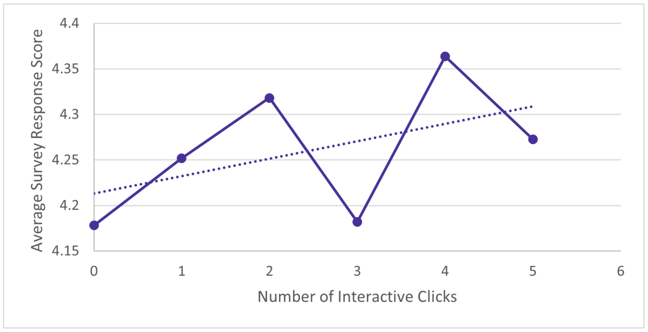


Figure 8: Average Likert survey score of users with different numbers of interactions with WorkedGen

as it helps me a lot with understanding the purpose of each code, for example the for loop was nicely explained when I clicked on the highlighted word 'for loop'." Users also liked the user interface of the keywords in the large explanations, as *"the use of boldened words in the descriptions of steps and problems makes the text much easier to comprehend."* Users liked how the LLM could be prompted to explain each line of code in more depth *"I also liked the feature that allowed us to click on individual lines of code for explanations as I found some terms unfamiliar."*

The ability to ask questions was generally well received. Some negative comments made were that the question input area was *"confusing"* and that *"the explanations only really worked in the context of the predetermined key words and weren't very helpful for other questions such as asking why the code used a particular technique as opposed to a different one."* Despite these flaws, most users found it very helpful and *"...enjoyed that you can ask specific questions about the code to help weed out any issue I have while wrapping my head around new topics."* One user commented that they appreciated having the opportunity to learn more about the question in the overview section before any code was shown: *"I liked how there was a section to ask about the question before even starting to attempt the actual code."* Of all the users who asked meaningful questions, only one user did not find it helpful.

5 DISCUSSION

5.1 How well can LLMs generate clear explanations in a specific knowledge domain?

The worked examples generated by LLMs were generally of a high quality. Under expert evaluation, 93% of generated examples on a range of novice to intermediate programming questions were seen to have clear explanations. The explanations were readable and succinct, and the code was written using good practices such as meaningful variable naming.

The key area where LLM-generated worked examples fell short of human-generated worked examples was in the reliance on prior knowledge. The LLM-generated explanations often relied on prior knowledge of programming concepts such as object-oriented programming (OOP), with which novice programmers are often not familiar. Multiple participants in the user study noted that they wanted the explanations to delve deeper into the decision-making process of programming, for example, in the selection between different forms of loops. Some of this can be remediated via the question-answer feature of WorkedGen, however, this feature was not built to model a chatbot, and did not allow for a sustained back-and-forth conversation.

These two weaknesses suggest that the worked examples would be of a higher quality if they contained more in-depth explanations. Future work could continue to explore effective prompting strategies, such as constraining the features used in the examples. WorkedGen was built to use a general, all-purpose prompt that would suit a wide range of questions and users. The disadvantage of the generality is that it does not account for the variety of background knowledge of the users.

5.2 How effectively can LLMs decompose a worked example into well-defined steps?

The prompt used to generate the worked example appeared to work well for decomposing steps. The LLM did not require a large amount of explanation on how to divide the steps, and almost all generated examples had between three to ten steps as instructed, following the method of constructing worked examples proposed by Atkinson et al. [2]. However, having a good number of steps does not always mean that the content is divided well. Some steps were so small as to be insignificant, which made the worked examples tedious to work through, especially as in WorkedGen, each step would be shown on a separate page. There were also cases where almost all code would be written in a single step. This can be overwhelming for users, and it means that users cannot benefit from fading, which has a studied effect on the users' cognitive load.

Overall, 71% of the generated worked examples were seen to have meaningful steps. These worked examples were all generated for use by novices, and so it would be an interesting area of further research to test more complex examples.

5.3 How do novices perceive the usefulness of on-demand LLM-generated worked examples?

The responses from the user study were generally positive. 77% of users found WorkedGen helpful, and 72% said that they would use the tool again. One student commented *"I will be using WorkedGen to help in my weekly lab exercises as well as a study tool for assignments and exams,"* and another said *"It was helpful for consolidating my knowledge so I will be able to use it for my revision and also when I'm stuck on a coding problem."* This aligns with previous research where students found LLM generated code explanations helpful for their learning [20, 28].

The main negative comments that WorkedGen received were that the explanations were sometimes too complex for novices. Worked examples are most effective when used to further a user's understanding of a concept that they are already familiar with [2]. For example, a worked example would be helpful to see how and when to use inheritance in an OOP course, but the students should first learn the theoretical basics of OOP. While LLMs are a helpful and powerful tool, they cannot replace the value of teachers and tutors. WorkedGen is best used to supplement content learned in class, not as a complete teaching tool.

5.4 How do users interact with worked examples, and how do they perceive the usefulness of these interactions?

The majority of interaction was with lines of code and by asking questions. The lack of clicks on keywords suggests that the explanations alone were in-depth enough for most users. The large number of clicks on LOCs is expected with novice users. Most questions asked were related to parts of the code, such as while loops. This makes sense as the users have not had a lot of programming experience, and so loops are still quite unfamiliar to them. This supports a previous finding, that the explanation should go more in-depth about the features of the code and any data structures that it uses. However, the LLM interaction allowed for these questions to be answered for the users. Users responded positively to this interaction, and Figure 8 shows that these interactions improved their overall experience of WorkedGen.

5.5 Limitations

There were a few limitations in this research. First, although WorkedGen was built to be used in any language, the vast majority of its uses were in Python. While the language should not have a large impact on its usefulness, it would still be interesting to test it in a range of languages. The user study was unsupervised. This was chosen to encourage more uses as it was easier for users to test WorkedGen in their own time, and it allowed for authentic data to be gathered. However, some students found the UI confusing and did not use WorkedGen to its full capabilities, so not as much interaction was observed in the study as there might have been with a supervised study.

6 CONCLUSION

This paper presents a novel tool, 'WorkedGen', purpose-built to employ a large language model to generate worked examples for novice programmers. This combats the high manual workload of creating worked examples and allows students to generate worked examples on-demand and interact with them for further explanations. WorkedGen uses the model GPT-3.5 Turbo and employs prompting strategies, including chaining and one-shot training to optimise the output of the LLM.

We make two primary contributions in this paper: the development of WorkedGen, and an empirical evaluation of the worked examples it generates through a user study in a first-year Python programming course and expert evaluation. This has shown that students find LLM-generated worked examples valuable, and additional interactive components allowed through an LLM are found to be especially useful. It also shows that LLMs can be used to generate effective and high-quality worked examples. WorkedGen can be improved through an updated and more intuitive user interface, and through prompt refinement to better explain coding principles with which novices may not be familiar. We propose that future work should be done to more thoroughly evaluate WorkedGen with a range of programming languages and more complex questions.

REFERENCES

[1] Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training LLMs for project-specific code-summarization. *arXiv preprint arXiv:2207.04237* (2022).

[2] Robert K Atkinson, Sharon J Derry, Alexander Renkl, and Donald Wortham. 2000. Learning from examples: Instructional principles from the worked examples research. *Review of educational research* 70, 2 (2000), 181–214.

[3] Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2022. Programming Is Hard–Or at Least It Used to Be: Educational Opportunities And Challenges of AI Code Generation. *arXiv preprint arXiv:2212.01020* (2022).

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D 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 Ziegler, Jeffrey Wu, Clemens Winter, Chris 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*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

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

[6] Paul Denny, Brett A Becker, Juho Leinonen, and James Prather. 2023. Chat Overflow: Artificially Intelligent Models for Computing Education-renAIssance or apocAIypse?. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1.* 3–4.

[7] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1.* 1136–1142.

[8] Paul Denny, Juho Leinonen, James Prather, Andrew Luxton-Reilly, Thezyrie Amarouche, Brett A Becker, and Brent N Reeves. 2023. Promptly: Using Prompt Problems to Teach Learners How to Effectively Utilize AI Code Generators. *arXiv preprint arXiv:2307.16364* (2023).

[9] Paul Denny, James Prather, Brett A. Becker, James Finnie-Ansley, Arto Hellas, Juho Leinonen, Andrew Luxton-Reilly, Brent N. Reeves, Eddie Antonio Santos, and Sami Sarsa. 2023. Computing Education in the Era of Generative AI. arXiv:2306.02608 [cs.CY]

[10] 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 Conference.* 10–19.

[11] James Finnie-Ansley, Paul Denny, Andrew Luxton-Reilly, Eddie Antonio Santos, James Prather, and Brett A. Becker. 2023. My AI Wants to Know If This Will Be on the Exam: Testing OpenAI's Codex on CS2 Programming Exercises. In *Proceedings of the 25th Australasian Computing Education Conference* (Melbourne, VIC, Australia) *(ACE '23)*. Association for Computing Machinery, New York, NY, USA, 97–104. https://doi.org/10.1145/3576123.3576134

[12] Mark Guzdial. 2015. What's the best way to teach computer science to beginners? *Commun. ACM* 58, 2 (2015), 12–13.

[13] Noor Hisham Jalani and Lai Chee Sern. 2015. The example-problem-based learning model: applying cognitive load theory. *Procedia-Social and Behavioral Sciences* 195 (2015), 872–880.

[14] Martin Jonsson and Jakob Tholander. 2022. Cracking the code: Co-coding with AI in creative programming education. In *Creativity and Cognition.* 5–14.

[15] Slava Kalyuga, Paul Chandler, Juhani Tuovinen, and John Sweller. 2001. When problem solving is superior to studying worked examples. *Journal of educational psychology* 93, 3 (2001), 579.

[16] Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J Ericson, David Weintrop, and Tovi Grossman. 2023. Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. *arXiv preprint arXiv:2302.07427* (2023).

[17] Anton Korinek. 2023. *Language models and cognitive automation for economic research.* Technical Report. National Bureau of Economic Research.

[18] Juho Leinonen, Paul Denny, Stephen MacNeil, Sami Sarsa, Seth Bernstein, Joanne Kim, Andrew Tran, and Arto Hellas. 2023. Comparing Code Explanations Created by Students and Large Language Models. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1* (Turku, Finland) *(ITiCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 124–130. https://doi.org/10.1145/3587102.3588785

[19] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, James Prather, and Brett A Becker. 2022. Using Large Language Models to Enhance Programming Error Messages. *arXiv preprint arXiv:2210.11630* (2022).

[20] 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.* 931–937.

[21] 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.* 37–39.

[22] Lauren E Margulieux, Briana B Morrison, and Adrienne Decker. 2020. Reducing withdrawal and failure rates in introductory programming with subgoal labeled worked examples. *International Journal of STEM Education* 7 (2020), 1–16.

[23] Roxana Moreno. 2006. When worked examples don't work: Is cognitive load theory at an impasse? *Learning and Instruction* 16, 2 (2006), 170–181.

[24] Kasia Muldner, Jay Jennings, and Veronica Chiarelli. 2022. A Review of Worked Examples in Programming Activities. *ACM Transactions on Computing Education* 23, 1 (2022), 1–35.

[25] Miranda C Parker, Leiny Garcia, Yvonne S Kao, Diana Franklin, Susan Krause, and Mark Warschauer. 2022. A Pair of ACES: An Analysis of Isomorphic Questions on an Elementary Computing Assessment. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1.* 2–14.

[26] Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. *arXiv preprint arXiv:2201.11227* (2022).

[27] James Prather, Paul Denny, Juho Leinonen, Brett A. Becker, Ibrahim Albluwi, Michelle Craig, Hieke Keuning, Natalie Kiesler, Tobias Kohn, Andrew Luxton-Reilly, Stephen MacNeil, Andrew Peterson, Raymond Pettit, Brent N. Reeves, and Jaromir Savelka. 2023. The Robots are Here: Navigating the Generative AI Revolution in Computing Education. arXiv:2310.00658 [cs.CY]

[28] 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 *Proceedings of the 28th International Conference on Intelligent User Interfaces.* 491–514.

[29] 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 Conference on International Computing Education Research-Volume 1.* 27–43.

[30] Rolf Schwonke, Alexander Renkl, Carmen Krieg, Jörg Wittwer, Vincent Aleven, and Ron Salden. 2009. The worked-example effect: Not an artefact of lousy control conditions. *Computers in human behavior* 25, 2 (2009), 258–266.

[31] Ben Skudder and Andrew Luxton-Reilly. 2014. Worked Examples in Computer Science. In *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148* (Auckland, New Zealand) *(ACE '14)*. Australian Computer Society, Inc., AUS, 59–64.

[32] John Sweller. 2006. The worked example effect and human cognition. *Learning and instruction* (2006).

[33] 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 conference on human factors in computing systems extended abstracts.* 1–7.

[34] Camilo Vieira, Junchao Yan, and Alejandra J Magana. 2015. Exploring design characteristics of worked examples to support programming and algorithm design. *Journal of Computational Science Education* 6, 1 (2015), 2–15.

[35] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382* (2023).

[36] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. 2022. Ai chains: Transparent and controllable human-ai interaction by chaining large language model prompts. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems.* 1–22.

[37] Rui Zhi, Thomas W Price, Samiha Marwan, Alexandra Milliken, Tiffany Barnes, and Min Chi. 2019. Exploring the impact of worked examples in a novice programming environment. In *Proceedings of the 50th acm technical symposium on computer science education.* 98–104.

[38] Daniel Zingaro and Leo Porter. 2015. Tracking student learning from class to exam using isomorphic questions. In *Proceedings of the 46th acm technical symposium on computer science education.* 356–361.