Ta et al. *Research and Practice in Technology Enhanced Learning* (2026) 21:25
https://doi.org/10.58459/rptel.2026.21025

Research and Practice in
Technology Enhanced Learning

RESEARCH Free and Open Access

Automatic generation of introductory programming exercises with large language models

Nguyen Binh Duong Ta *, Hua Gia Phuc Nguyen and Swapna Gottipati

*Correspondence:
donta@smu.edu.sg
School of Computing and Information Systems, Singapore Management University,
80 Stamford Rd,
Singapore 178902
Full list of author information is available at the end of the article

Abstract

Despite recent advances in code generation made possible by large language models (LLMs), programming is still an essential skill that computing students need to master now and in the foreseeable future. In learning programming, frequent practices with exercises set at an appropriate difficulty and knowledge level is of crucial importance for students. However, it's not a trivial task for instructors to create many good quality exercises customized for each student. Programming problems found on Internet sources such as LeetCode are mostly too challenging for novice programmers with no prior coding knowledge. Recent work in AI-enabled education has been leveraging LLMs for adaptive feedback generation on code submitted by students. Not much work has been done in generating customized exercises for students to have more practice. In this work, we propose ExGen, an automatic exercise generation system which uses LLMs such as OpenAI's GPT models to generate on-demand, customized, and ready-to-use programming exercises for individual students. ExGen is designed as a plugin to Visual Studio Code. It incorporates a set of prompting strategies for candidate exercise generation, and a novel chain of automatic filtering mechanisms to select ready-to-use exercises. ExGen is convenient to use as compared to chatbots such as ChatGPT. We have conducted an extensive performance evaluation using more than 1400 generated Python exercises. We considered several prompting strategies with various keyword and seed exercise types, filtering techniques, difficulty levels, and LLMs with different generative performance and cost. The results demonstrated the effectiveness of ExGen's design and implementation.

Keywords: Programming courses, Ready-to-use exercise generation, LLMs, Prompt engineering, Auto-filtering, Difficulty levels

Introduction

Recent developments in pre-trained large language models (LLMs), e.g., OpenAI GPT models, Google Gemini, Anthropic Claude, etc., have enabled impressive performance in automatic generation of code (Liu et al., 2024; Tipirneni et al., 2024). However, that does not mean that we should stop training new programmers altogether. One reason is that the correctness of automatically synthesized code by LLMs, especially for real-world software development scenarios, is still very much a concern (Du et al., 2024). It is more realistic to say that well-trained programmers using AI for code generation will replace those who do not, rather than AI will completely replace programmers. In this context, programming is still an essential skill that computing students must master in the foreseeable future (Becker et al., 2023).

It is well-known that computer programming is a challenging subject for many new university students (Keuning et al., 2018). To learn effectively, students must practice on their own frequently with suitable exercises designed for their level of programming skills and knowledge. However, it is usually impractical for instructors to manually create enough exercises for all students, let alone creating customized or personalized exercises for each individual student on-demand, i.e., when they need to practice. Exercises from Internet platforms for coding practices and interview preparations (Joshi et al., 2023) such as LeetCode.com might not be ready-to-use in introductory courses as they are not customized for the skill levels of new computing students. Many easy-level LeetCode problems can be quite challenging for novice programmers (Ta et al., 2022). Intermediate or harder problems found online often require advanced data structures and algorithm concepts not taught in an introductory course. Such a course usually assumes no prior coding background from students.

Automatic generation of programming exercises has been an important problem in technology-enabled education (Zavala & Mendoza, 2018). Recently, the application of pre-trained LLMs in supporting exercise and feedback generation has been gaining attention. Sarsa et al. (2022) explored OpenAI Codex for the purpose of creating new Python exercises and code explanations. They provided a comprehensive evaluation of the output generated by Codex via manual inspection, code compilation, and unit testing. They showed that a large percentage of the generated exercises were sensible and new, but most of them were not directly usable by students, i.e., not ready-to-use. More recently, Del Carpio Gutierrez et al. (2024) examined GPT-4's capabilities in generating contextualized programming exercises. However, both works did not address the problem of on-demand generation of ready-to-use exercises customized for various levels of difficulty. Becker et al. (2023), Finnie-Ansley et al. (2023) and Kazemitabaar et al. (2023, 2024) discussed and evaluated the use of LLMs to generate sample solutions and to provide adaptive feedback on code submitted by students, but not automatic exercise generation. More recent LLM-

based works regarding programming, e.g., Du et al. (2024), Tipirneni et al. (2024), etc., focused on the quality of automatically generated code for manually defined coding problems.

In this work, we propose ExGen, a software tool for automatically generating new, ready-to-use exercises for individual students learning introductory programming in Python. ExGen makes use of LLMs which have been pre-trained on a massive amount of text and code such as OpenAI's GPT-3.5 Turbo and GPT-4 Turbo. In ExGen, we implement carefully designed LLM prompting strategies, automatic filtering of exercises, and seamless integration with Visual Studio (VS) Code. ExGen makes it convenient for students to work on new exercises that are customized for their skill levels. An innovative component of ExGen is that it makes use of LLMs not just to generate new exercises, but also to automatically check if the newly generated exercises are ready-to-use or not. In this way, ExGen is different from existing research that investigated exercise generation based on pre-defined templates (Zavala & Mendoza, 2018), or required manual checking to verify the quality of the generated exercises (Del Carpio Gutierrez et al., 2024).

We make the following contributions in this paper:

- We considered the problem of auto-generating ready-to-use exercises customized for a certain difficulty level, i.e., "easy", "intermediate", and "hard". The exercises are to be generated on-demand, i.e., upon requests from individual students. In this problem, the ready-to-use exercises are defined based on the clarity of its problem statement, the correctness of its reference solution and test cases, and the required difficulty levels.
- We designed "zero-shot" and "few-shot" prompting strategies to obtain appropriate responses from the pre-trained LLMs. In each prompt, the difficulty levels are specifically defined; suitable example exercises, i.e., shots, are provided; and various keywords are supplied to generate exercises which can cover different concepts or realistic application domains.
- We proposed a mechanism of chaining different filtering checks which auto-select ready-to-use exercises instead of requiring human experts to inspect each of the LLM output and make decisions manually. The checks are based on popular software engineering techniques, namely code compilation and unit testing, combined with LLM-based difficulty classification techniques.
- We implemented a fully functional VS Code extension which makes it more straightforward for students to generate new exercises and practice coding at their own pace. We noted that using existing web interfaces such as ChatGPT or OpenAI's Playground requires a lot of copy/paste actions for adjusting prompts and submitting requests, which are not convenient when doing programming.

- We conducted an extensive evaluation of ExGen using more than 1400 programming exercises generated via the latest version (at the time of experiments) of the GPT models offered as web services by OpenAI. To evaluate our auto-filtering checks, each of these exercises was manually inspected to see if they were ready-to-use; and compared to ExGen's decision. The result demonstrated the effectiveness of ExGen and the proposed solution design.
- We considered the impacts on exercise generation performance of three different types of keywords, and two different types of example exercises used as shots in the LLM prompting strategies. We noted that appropriate keywords have a significant impact on the performance of ExGen when generating exercises for a given difficulty level.
- We compared the exercise generation performance between the faster and much cheaper GPT-3.5 Turbo model with the state-of-the-art (at the time of experiments) and more expensive GPT-4 Turbo model. This is because the cost of using a pay-per-use AI service like OpenAI can be significant in the long run, especially when we have large numbers of students. In addition, the more powerful AI models might require more time for inference, so they may run much slower. This would affect the ability of ExGen to generate new exercises in a truly "on-demand" manner. The results provided interesting insights and implications on whether we need to use the latest and more powerful AI models in every situation.

The rest of this paper is organized as follows. Section "Problem statement" defines the automatic exercise generation problem in introductory programming courses. Section "ExGen: design and implementation" describes ExGen's design and implementation, including our prompting and filtering approaches. Section "Evaluation" presents the evaluation methodology, result analysis, and a discussion on possible biases and threats to the validity of this study. Section "Related work" concludes the paper and outlines future directions.

Problem statement

We are interested in auto-generating ready-to-use exercises, i.e., those that a student can work on right away without further modifications by human experts. To be considered ready-to-use, a generated exercise would need to have: 1) a clear problem statement, 2) working solutions and test cases, and 3) an appropriate difficulty level as requested by the student. This is a challenging problem, as Sarsa et al. (2022) found that most generated exercises were not ready-to-use when using OpenAI Codex. Another previous evaluation

(Ta et al., 2023) with GPT-3.5 Turbo demonstrated that it is not trivial at all to generate ready-to-use exercises at higher difficulty levels. Del Carpio Gutierrez et al. (2024) showed that GPT-4 could perform well when generating new programming exercises; however, it was not clear how difficult their generated exercises were.

On top of generating clear, well-defined problem statements, producing exercises with the right level of difficulty is important to students learning programming. For instance, beginners would be encouraged by easy exercises covering the concepts taught in class as they are just getting started. On the other hand, above average students would like to be challenged with harder problems requiring a certain level of higher order thinking. Many programming problems available on Internet platforms (Joshi et al., 2023) are not appropriate for students in our course due to them: 1) being very difficult, and/or 2) requiring advanced concepts such as dynamic programming, tree-based data structures, in-depth complexity analysis, etc., which are not covered in an introductory course for first-year students.

In this work, we define three levels of difficulty for exercises which have been used in an introductory programming course at our university. The three levels of difficulty, namely "easy", "intermediate", and "hard", are defined relative to each other. These levels can cater to a cohort of students with wide-ranging levels of skills and abilities. As the course is more about basic programming and computational thinking skills, and less about using Python, some basic exercises can be made harder by disallowing certain Python utilities such as sorting functions. We can also impose additional constraints on the number of loops due to efficiency reasons, and as a result, the difficulty is increased. In the following three examples, we explain the difficulty levels for each exercise which has been used in our course.

Example 1 (Difficulty Level: Easy)

Problem statement:

```
Write a Python function called all_older_than(). The function takes two
parameters: (1) A list of integers called age_list, where each element
indicates the age of a person. (2) An integer called n, which is a
threshold. The function returns True if ALL the age values in age_list
are larger than n, and False otherwise. For example,
all_older_than([24, 36, 45, 21], 20) returns True, and
all_older_than([24, 36, 45, 21], 23) returns False. If age_list is
empty, the function returns True.
```

Test cases:

```
assert all_older_than([24, 36, 45, 21], 20) == True
assert all_older_than([24, 36, 45, 21], 23) == False
```

Reference solution:

```
def all_older_than(age_list, n):
    for age in age_list:
        if age <= n:
            return False
    return True
```

Fig. 1 Example of an "easy" level programming exercise

Example 1: This is an "easy" exercise which tests basic list usage skills using a "for" loop and simple conditional statements in Python. Most students, including those who are new to programming, after being taught the concepts, can handle this exercise. We have been observing that a few might need help with the correct placement of the "return" statement. Note that the "easy" exercises are more than just trivial programming tasks such as creating a list, adding two numbers, etc. In this example, we provide a reference solution, which is quite straightforward; and two test cases which are used to validate the code implemented by students.

Example 2: an "intermediate" exercise for string and list manipulation. The number of lines of code as well as the logic for a possible solution to this exercise is more complicated than that of Example 1. The student would have to figure out the logic with two different "for" loops and relevant conditional statements for each loop. A common logic mistake for programming novices that we have observed is that they only check if any string in the 1st list is not in the 2nd list and append that string for returning later. Therefore, in the context of this paper, Example 2 is more difficult than Example 1.

Example 3: a "hard" exercise for string manipulation. This exercise is not as straightforward as Example 1 or 2, as it requires students to first understand how the Caesar cipher works. Then, students are required to make use of the "ord" function, the key, and the "modulo" operator to compute the encrypted character for each plaintext character, given the size of the alphabet. Most novice programmers would struggle with this exercise. In this work, Example 3 is considered more difficult than Example 2.

Example 2 (Difficulty Level: Intermediate)

Problem statement:

```
Write a Python function called get_non_common_strings(). The function
takes in two lists of strings, str_list1 and str_list2. The function
returns a list of strings that can be found in either str_list1 or
str_list2, BUT NOT in both. For example, if str_list1 is ["a", "b",
"c", "d"], str_list2 is ["b", "d", "e", "f"], then this function
returns ["a", "c", "e", "f"].
```

Test cases:

```
assert get_non_common_strings(["a", "b", "c", "d"], ["b", "d", "e",
"f"]) == ['a', 'c', 'e', 'f']
assert get_non_common_strings(["a", "b", "c", "d"], ["b", "d", "c"]) ==
['a']
```

Reference solution:

```
def get_non_common_strings(str_list1, str_list2):
  result_list = []
  for str1 in str_list1:
    if str1 not in str_list2:
      result_list.append(str1)
  for str2 in str_list2:
    if str2 not in str_list1:
      result_list.append(str2)

  return result_list
```

Fig. 2 Example of an "intermediate" level programming exercise

Example 3 (Difficulty Level: Hard)

Problem statement:

```
Write a Python function titled encode_caesar that takes in two
parameters namely plaintext and key, encrypts the plaintext and then
returns the ciphertext (which has been encoded using the key). The
plaintext is the message to be encrypted to ciphertext, and key is the
shift (which can be right or left). A key value of 3 indicates that
each character in the plaintext should be shifted right by 3, and key
value of -3 indicates that each character in the plaintext should be
shifted left by 3.
```

Test cases:

```
assert encode_caesar("If you want something badly enough, do not give
up!", -3) == "Fc vlr txkq pljbqefkd yxaiv bklrde, al klq dfsb rm!"

assert encode_caesar("Programming is SO FUN!", 12) == "Bdasdmyyuzs ue
EA RGZ!"
```

Reference solution:

```
import string
def encode_caesar(plaintext, key):
  ciphertext = ''
  for i in range(len(plaintext)):
    if plaintext[i] in string.ascii_lowercase:
        ciphertext += string.ascii_lowercase[(ord(plaintext[i])-
ord('a')+key)%26]
    elif plaintext[i] in string.ascii_uppercase:
        ciphertext += string.ascii_uppercase[(ord(plaintext[i])-
ord('A')+key)%26]
    else:
        ciphertext += plaintext[i]
  return ciphertext
```

Fig. 3 Example of a “hard” level programming exercise

We note that it is not practical to manually create a lot of such ready-to-use exercises on-demand, i.e., when students need them. Furthermore, the new exercises should be personalized to individual skill levels and learning concepts, as many students might be working concurrently on different kinds of problems at various difficulty levels. In the below section, we describe our approach to realizing automatic generation of ready-to-use exercises.

ExGen: design and implementation

Figure 4 illustrates the interactions between students, instructors and ExGen in a typical introductory programming course. As ExGen aims to minimize instructor’s involvement outside the classrooms, the only role of the instructor is to provide a limited number of sample (seed) exercises with three difficulty levels: “easy”, “intermediate”, and “hard”. Each student in the course can download these exercises into a personalized database which is saved locally. Students can load any exercises from their own database, work on them, and use ExGen to generate more exercises to practice. Students can choose to save new exercises into their personalized database for reference later. The exercises are saved with

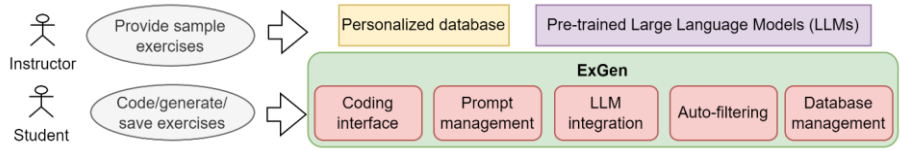

Fig. 4 Design and implementation of ExGen

a corresponding difficulty level; and can be used as input (i.e., examples) for generating more exercises later.

ExGen provides functionalities for managing a personalized database of exercises for each student, generating new exercises via pre-trained LLMs, and providing a suite of auto-filtering checks to give students ready-to-use exercises at a difficulty level suitable for their knowledge and programming skills. In the below, we provide a brief description of each ExGen component as shown in Figure 4, and relevant implementation details.

Prompt management and LLM integration

ExGen generates new exercises by first constructing a "prompt" as a query to be sent to a pre-trained LLM such as OpenAI GPT-3.5 Turbo. It obtains a list of candidate exercises for further filtering before presenting a few selected ones in an IDE, e.g., VS Code, for the student to work on. Constructing good prompts for LLMs in many application domains, i.e., "prompt engineering", is an active area of research (Arawjo et al., 2024; Liu et al., 2023). LLM prompts may consist of questions, instructions, examples, etc., so that the model would reply with an output which has some desired qualities and/or quantities. In "zero-shot" prompting, we can query the model without providing any example of expected results. On the other hand, "few-shot" prompts provide several actual examples to the model.

In the GPT-3.5 and GPT-4 Chat Completions API, which are used in ExGen's implementation, input prompts can be given as a series of messages with different parameters. Each message is an object with a role, which is either "system", "user", or "assistant", together with the content of the message. Usually, the first message would be a "system" message which helps define the behavior of the AI assistant. We can then alternate between "user" and "assistant" messages, in which "user" messages provide specific instructions for the AI assistant, and "assistant" messages can be used to provide examples of desired outputs, i.e., the "shots" in few-shot prompting. All messages must be constructed and passed to the LLM in a single API call, so that the model would have the right context when generating the new exercises. Note that OpenAI APIs are stateless, i.e.,

they do not maintain the memory of past requests. In this work, we consider two different strategies of prompting for exercise generation using OpenAI API, namely:

Zero-shot Prompting (referred to as "zero-shot"): provides keywords on the programming concepts and difficulty levels required by a student, without using any examples of exercises. The prompt starts with a system message, e.g., "You are a helpful teaching assistant for undergraduates who are learning introductory programming in Python. You need to generate Python exercises for students to practice.". In the system message, we also define the required difficulty levels, i.e., explaining what it means for an exercise to be considered as "easy" or "hard", as shown in Message 1 of Figure 5. Finally, we add a user message, e.g., "Give me three easy Python exercises with this keyword: house".

Few-shot Prompting (referred to as "few-shot"): provides a few actual examples on how the desired output should look like. In essence, when requested by students, ExGen constructs a series of messages following the GPT Chat Completions model as shown in Figure 5. ExGen first inspects the current exercise that the student is working on. Second, it searches for other exercises on the same concept, e.g., string, list, etc. and having a similar

```
Message 1: {"role": "system", "content": "You are a helpful teaching
assistant for undergraduates who are learning introductory programming
in Python. You need to generate Python exercises for students to
practice. There are three levels of difficulty for the exercises:

Easy: most students will solve the problem quickly with a few lines of
        code.

Intermediate: most students will take more time to solve the problem, and
        they need to write more code. Many students, but not all, will
        be able to solve the problem in the end.

Hard: most students will take a lot of time to solve the problem. Many
        of them will not be able to solve the problem in the end."}

Message 2: {"role": "user", "content": "Give me a {difficulty level}
Python exercise."}

Message 3: {"role": "assistant", "content": "Here is one {difficulty
level} Python exercise: {example of a relevant exercise from the
database}"}

Message 4: {"role": "user", "content": "Good. I want one more
{difficulty level} Python exercise. Print the result with the same
format as the previous ones."}

... (more messages for remaining examples)

Last message: {"role": "user", "content": "Good. I want {N} more
{difficulty level} Python exercises using this keyword: {keyword}.
Print the result with the same format as the previous ones."}
```

Fig. 5 Messages for Few-shot Prompting

level of difficulty in the student's database. ExGen then sequentially appends the relevant examples, each with a problem statement, solution, and test cases, with the appropriate roles into the list of messages. The result will be like a conversation in which the LLM plays the role of a teaching assistant guided via a multi-turn conversation with several concrete examples. The last message is a user message requesting the LLM to generate N new exercises.

In the implementation of "few-shot", ExGen uses examples ranging from 1 to 3 shots with the default parameters of the GPT-3.5/4 Turbo model offered by OpenAI. The output of this stage is N candidate exercises, each having: 1) a problem statement, 2) a solution, and 3) test cases. ExGen will conduct auto-filtering for these candidate exercises in the next stage.

Prompting keywords and few-shot examples

ExGen allows students and instructors to specify their own keywords to create more varied exercises with the desirable programming concepts and practical, real-world context; as shown in Figure 5 (the last message). We hypothesize that different types of keywords might have different impacts on exercise generation performance. In this work, we consider three different types of keywords for exercise generation, namely "simple", "complex", and "domain".

- The "simple" keywords are about basic programming concepts like "lists", "strings", etc. The LLM is supposed to generate exercises covering the specified concept when given one of these keywords.
- The "complex" keywords cover more involved and sophisticated programming concepts, e.g., "nested lists", "nested loops", etc. It is expected that these keywords would help in generating more challenging exercises, i.e., "intermediate" or "hard".
- The "domain" keywords are used so that the generated exercises could have more practical and interesting contexts. Examples of such keywords include "food delivery", "housing", "table allocation", etc.

In addition to keywords, the types of examples exercises provided as "shots" to the "few-shot" prompting strategy may have some impact on the quality of the generated exercises. Therefore, we consider two different types of example exercises to be used as "shots":

- The "concept" exercises concern basic skills in handling various programming concepts such as for/while loops, string indexing, list manipulations, using dictionaries, etc. For example, the exercise can ask students to find a number with a certain value in a given list.

- The "domain" exercises are about real-world, practical problems that can be solved by programming. For example, the exercise can ask students to write a function to assign applicants to rental flats.

We provide more detailed examples of different keyword and example exercise types; and evaluate their impact on exercise generation performance in Section "Evaluation" of this paper.

Auto-filtering of generated exercises

In ExGen, we implement several methods to filter exercises generated by LLMs. The methods are chained, i.e., a newly generated exercise must successfully clear the previous filter before it is checked by the next filter, as shown in Figure 6. We consider standard software engineering techniques as well as AI-based approaches for filtering. First, ExGen will check if the Python solution code can be compiled to bytecode without any errors. Then it will run all the unit test cases which are generated by the LLM together with every new exercise.

After a new exercise can pass the first two filters, ExGen will do an additional check regarding the difficulty level of the exercise. The rationale for implementing this final check is that LLMs such as the OpenAI's GPT models are probabilistic in nature, and their answers might be different from one interaction to the next. Therefore, it could be beneficial to ask an LLM to verify or reflect on its own answers generated previously. This approach is one of the key strategies in agentic design patterns (Koutcheme et al., 2024; Wu et al., 2023), which have become popular recently in constructing AI-enabled applications and services using pre-trained AI models.

In our work, for the difficulty check, ExGen will perform a classification task by first constructing an appropriate prompt as shown in Figure 7 and then asking the LLM to classify the generated exercise in terms of difficulty levels. The prompt has two examples for each level of difficulty in a series of user and assistant messages. For the last user message, we ask the LLM to provide a classification for a new candidate exercise based on the provided examples. If the difficulty of the new exercise matches the required difficulty level, it will be shown to the student as a ready-to-use exercise using ExGen's VS Code extension.

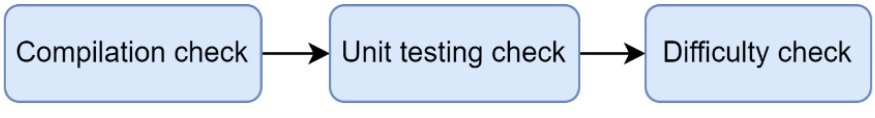

Fig. 6 Chain of filters in ExGen

```
Message 1: {"role": "system", "content": "You are a classification
model that will classify the difficulty of Python exercises. There are
three levels of difficulty for the exercises:
1: This is for easy problems. Most students will solve the problem quickly
         with a few lines of code.

2: This is for intermediate problems. Most students will take more time
         to solve the problem, and they need to write more code. Many
         students, but not all, will be able to solve the problem in the
         end.

3: This is for hard problems. Most students will take a lot of time to
solve the problem. Many of them will not be able to solve the problem
in the end."}

Message 2: {"role": "user", "content": "I want to you classify this
exercise: {example exercise}"}

Message 3: {"role": "assistant", "content": "Difficulty: " {example's
difficulty level}}

... (more messages for remaining examples)

Last message: {"role": "user", "content": "I want you to classify this
exercise: {candidate exercise}"}
```

Fig. 7 Prompt for difficulty classification

We should note that the difficulty check as implemented in Figure 7 can be costly considering the token pricing of OpenAI's LLMs. The classification prompt needs to include example exercises of each difficulty level, which results in a significant number of tokens used in each check. However, our chained filtering mechanism could reduce the cost, i.e., as only exercises which passed compilation and unit testing checks will be considered in the difficulty check.

Coding interface and database management

ExGen provides a convenient interface for students to work and generate exercises right in VS Code. We have used the ChatGPT web app, as well as OpenAI's Chat Playground extensively, and we observed that it takes a lot of time and effort to copy/paste prompts and exercises for generation tasks in these apps. Sample screenshots of ExGen's interface are shown in Figures 8a to 8d. Students can click on "+ More exercise" at the right bottom of the IDE to generate new ready-to-use exercises (Figure 8a). They can then enter the keyword to be used (Figure 8a), specify a desired difficulty level ("easy" to "hard", Figure 8b), select the prompting strategy (zero to three shots, Figure 8c), and even some filtering options (whether they want to apply unit testing and/or difficulty check, Figure 8d).

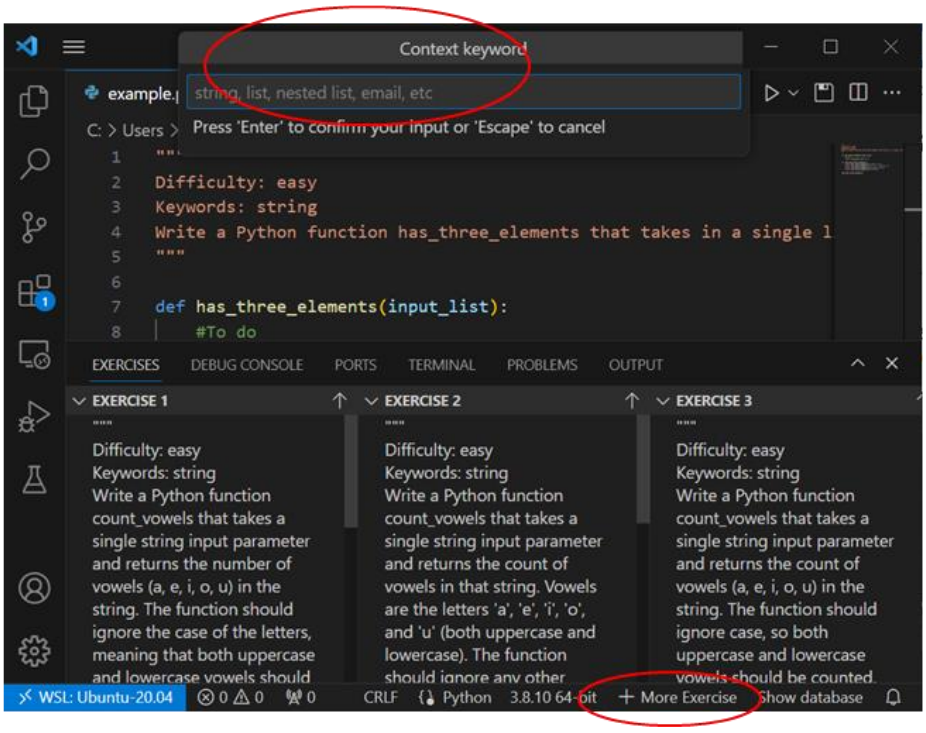

Fig. 8a ExGen's VS Code Interface (a)

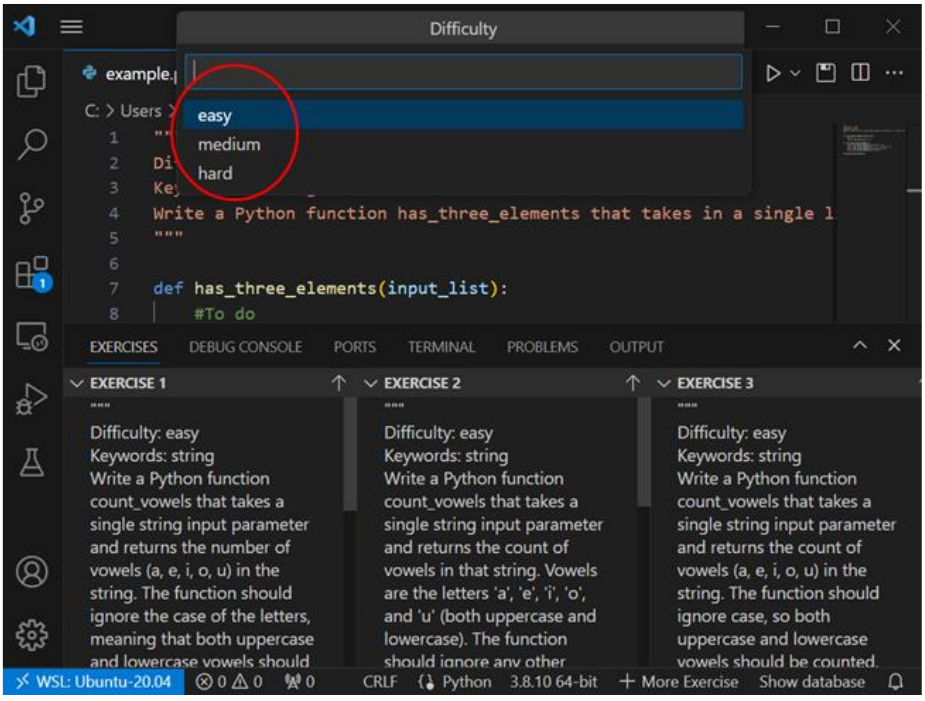

Fig. 8b ExGen's VS Code Interface (b)

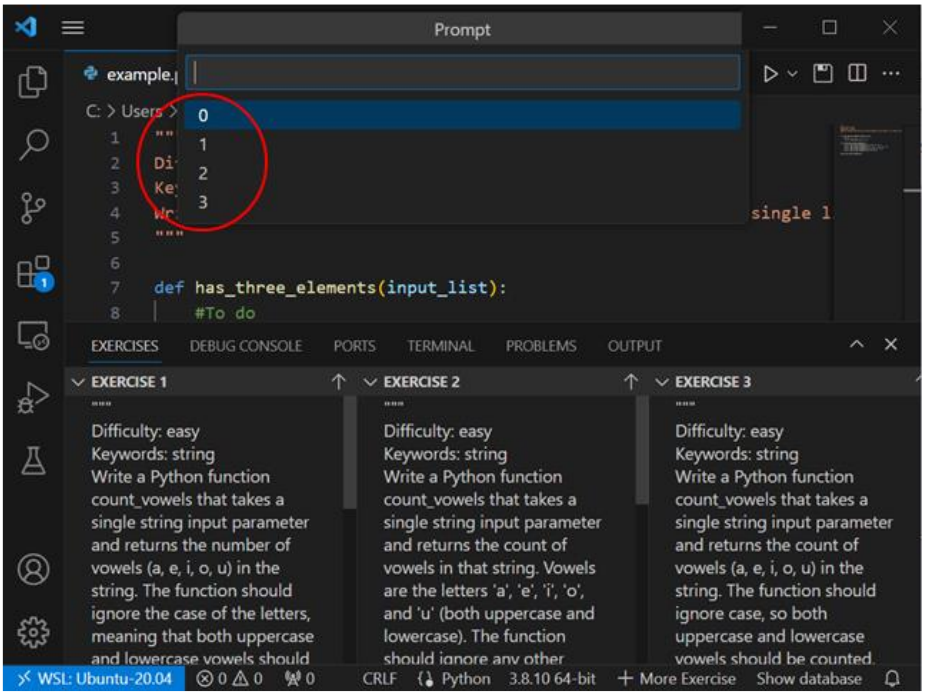

Fig. 8c ExGen's VS Code Interface (c)

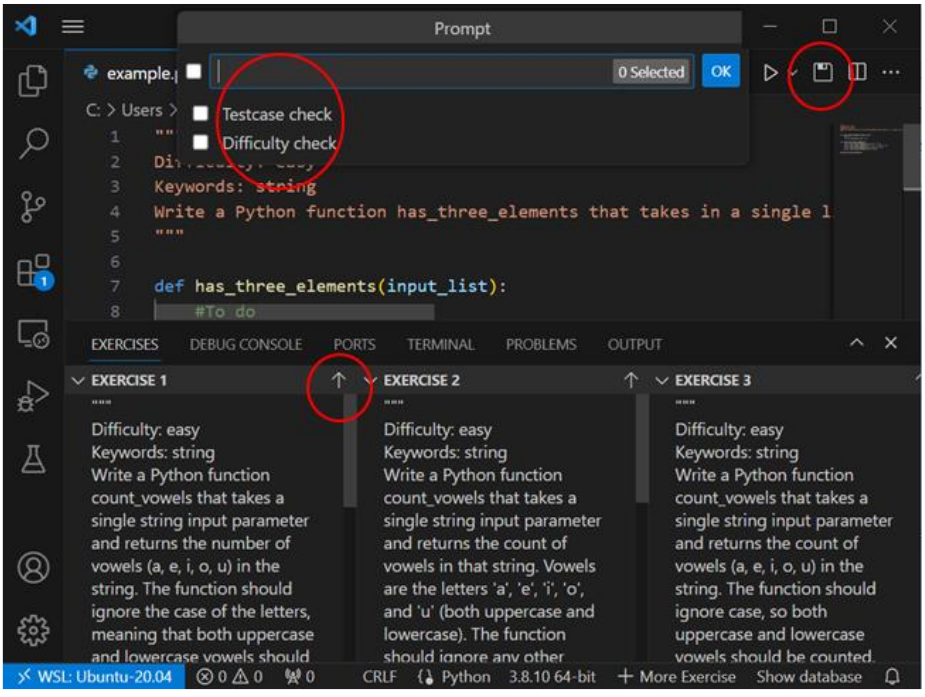

Fig. 8d ExGen's VS Code Interface (d)

The filtering options provide some flexibility for users. If all filters are selected, ExGen would return the most ready-to-use exercises possible, but it might take more time and cost, especially if we want to generate "hard" exercises. On the other hand, users can choose to go for a quick generation (and manually inspect or modify the result later to suit their needs) by not applying the filters. The newly generated and filtered exercises are shown in several text boxes at the bottom of the VS Code UI. Students can work on each of them right away by clicking the arrow (red circled) located at the top right corner of each text box, as shown in Figure 8d.

Students also have the option to change the problem statement, solutions or test cases used as examples in the prompt before clicking "Submit". ExGen will automatically connect to an LLM API, generate, and filter the exercises. Students can also save selected ready-to-use exercises into his/her own personalized database. To make it easy for any users, we use the TinyDB package, which is a portable Python document-oriented database, instead of setting up a fully functional SQL database engine like MySQL. The database comes bundled in ExGen so students can just use it right away without further setup or installation.

Evaluation

In this section, we aim to answer the following research questions (RQs):

- **RQ-1**: Which is the best prompting strategy for generating ready-to-use exercises?
- **RQ-2**: How effective is the auto-filtering approach implemented in ExGen?
- **RQ-3**: How do different types of keywords used in the LLM prompts impact generation performance?
- **RQ-4**: How do different types of example exercises used as shots in the LLM prompts impact generation performance?
- **RQ-5**: How does the latest and more expensive LLM (GPT-4 Turbo) perform when compared to the much cheaper GPT-3.5 Turbo in ready-to-use exercise generation?

Dataset and performance measures

We used a seed set that has a total of 59 manually crafted Python exercises including all levels of difficulty to be used as examples (shots) in ExGen's "few-shot" prompt. To quantify the impact of different types of examples on ExGen's generation performance, we further divided the seed set into two different smaller sets, namely "concept" exercises and "domain" exercises, as shown in Table 1. The "concept" exercises concern basic skills in handling various programming concepts such as for/while loops, string indexing, etc. On the other hand, the "domain" exercises are about real-world, practical problems that can be solved by programming. Table 1 shows two example excerpts of such exercises.

Table 1 Statistics and examples of the exercises (used as shots) from the "concept" and "domain" datasets

Dataset	No. of exercises	"easy"	"intermediate"	"hard"	examples (not a full problem statement)
"concept" exercises	28	7	12	9	Write a Python function containsDuplicate(nums) where given an integer list nums, return true if any value appears at least twice in the list, and return false if every element is distinct.
"domain" exercises	31	11	11	9	Write a Python function get_nearest_driver that takes in the following two parameters: a tuple containing the passenger's location in the form of (latitude,longitude), and a dictionary containing each driver's name and his/her current location in the form of a tuple (latitude,longitude).

In addition to the example, we need to provide keywords as part of the LLM prompt to generate new and diverse exercises. This is necessary for both "zero-shot" and "few-shot" prompting strategies. In this work, we studied the impacts of three different types of keywords on exercise generation performance, namely "simple", "complex", and "domain". The "simple" keywords are about basic programming concepts like lists, strings, etc. The "complex" keywords cover more involved and sophisticated programming concepts, e.g., nested lists, nested loops, etc. The "domain" keywords, similar to "domain" exercises, are used so that the generated exercises could have more practical and interesting contexts. Table 2 shows all the keywords used in our experiments.

Table 2 Different types of keywords used in the experiments, namely "simple", "complex", and "context"

Simple	"string", "list", "dictionary", "tuple", "boolean", "functions", "data types", "relational operations", "for loop", "while loop", "conditional statements", "arithmetic operations", "logical operations"
Complex	"nested lists", "nested for loops", "a combination of conditional statements and loops", "string slicing with nested loops", "dictionary of lists and strings", "combination of for and while loops", "list indices and for loops", "nested conditional statements", "list traversing and computation of each element in nested loops", "manipulating multiple lists", "processing strings, lists and dictionaries", "comparing indexes in lists, strings, or tuples"
Domain	"food and drinks", "housing allocation", "electronic devices", "travel planning", "driving instructions", "swimming training", "football games", "election and voting", "online shopping", "shipping and delivery", "table allocation", "students and teachers", "names and addresses", "email and messages"

We generated a total of 1404 exercises (468 for each difficulty level) using ExGen via the API of GPT-3.5 Turbo (version 0125), and GPT-4 Turbo (version 0125-preview). GPT versions were the latest at the time of experiments. By default, we used GPT-3.5 Turbo unless stated otherwise. For OpenAI pre-trained LLMs, we can choose our own hyperparameters which could influence the output of the models. We set "temperature", which is the parameter to control the randomness of the output to 0.7 (larger values result in more randomness), "max_tokens", which indicate the number of maximum tokens to be generated is 4095 and left other hyperparameters at their default values. For "few-shot" prompting, a default number of 3 examples was used in the experiments. The users of ExGen including instructors and students do not have to choose these hyperparameters.

We then manually inspected each of these exercises to determine if they are ready-to-use. A ready-to-use exercise must satisfy the following criteria used in the manual inspection: 1) a clear problem definition, 2) a correct solution and test cases, and 3) a correct difficulty level as requested in the prompt. For a given exercise generated by ExGen, the manual inspection process is as follows:

- Step 1: The inspector checks if the solution and the test cases of the exercise work properly, i.e., all test cases can pass. If not, the exercise will be labelled as NOT ready-to-use immediately. Otherwise, the inspector moves to the next step.
- Step 2: The inspector reads the problem definition and checks for correctness, clarity and consistency. The inspector then looks at the solution and test cases to confirm if they are indeed correct given the problem definition. If not, the exercise will be labelled NOT ready-to-use. Otherwise, the inspector moves to the next step.
- Step 3: The inspector then checks the difficulty of the given exercise. To ensure that the difficulty level given to an exercise is valid, the inspector compares it to the sample exercises described in Table 1 using a triangulation procedure. For instance:
    - A newly generated "intermediate" exercise should be harder than an existing "easy" one. For example, it should require more time to understand the problem definition, more lines of code to be written, and more than just a simple loop to solve.
    - An "intermediate" exercise should be easier than an existing "hard" one. A "hard" exercise should have a more complicated problem definition, require many more lines of code and the application of more difficult programming techniques such as dynamic indexing into nested lists or strings, etc.

    - If the difficulty level is not appropriate, the exercise will be labelled NOT "ready-to-use", even though its problem definition, solution and test cases are valid.
    - The inspector completes the inspection of all the exercises generated for a given difficulty level, e.g., "easy". Then the inspector moves on to inspect exercises of another difficulty level, i.e., "intermediate", then "hard".
- Step 4: after the inspection is completed for all the generated exercises, the inspector reviews the results. Using a triangulation procedure as described above, the inspector can determine if an exercise should be labelled differently. This step is repeated until there is no more change in the labels of all the exercises.
- To avoid inconsistencies between multiple human evaluators, the manual inspection was done by a single, highly experienced instructor with more than 10 years in the subject of programming (the 1st author of this paper). The manual inspection of all the exercises generated in this paper took about a month to complete.
- Although in our experiments, ExGen requested that 3 different exercises be generated each time, in several cases there could be duplications, i.e., identical exercises provided as responses to a generation request. We only counted the unique exercises which passed manual inspection as ready-to-use.

We then measured the performance, i.e., accuracy, of each prompting strategy and filtering check when compared to the manual inspection results. For example, when "few-shot" prompting produces 5 ready-to-use out of 10 generated exercises, the performance of "few-shot" is calculated as 50%. On the other hand, the performance of a filtering check is calculated by how often it agrees with the manual inspection result. For example, the performance of the unit testing check is calculated as 70% if it provides the same result, i.e., ready-to-use or not, as the manual inspection for 7 out of 10 generated exercises. We measured the individual percentage agreement of each filtering check, as well as the overall filtering performance.

We also measured the average cost incurred per ready-to-use exercise, which is calculated by taking the total token cost (in $) for using the LLM and divided by the number of ready-to-use exercises for a given difficulty level and prompting strategy. We tracked the total number of tokens used for both requests and responses, and calculated the cost using OpenAI's current pricing model, which is available at https://openai.com/api/pricing. Similarly, the average generation time per ready-to-use exercise is calculated by dividing the total amount of time (in seconds) to generate all exercises for a given difficulty level and prompting strategy to the number of ready-to-use exercises obtained.

Results and discussion

RQ1-Comparing different prompting strategies

Table 3 shows the accuracies, i.e., the percentages of ready-to-use exercises, the average costs (in $) and the average times (in seconds) per ready-to-use exercise for the two prompting strategies and three difficulty levels. These are the combined results for all three types of keywords and two types of seed exercises used as examples in the prompts. We observed that the "few-shot" prompting performed better than "zero-shot" prompting in all cases. This is because "few-shot" incorporates relevant examples (programming exercises) to better guide the LLM in generating new exercises with desired levels of difficulty.

In addition, there were a few other observations regarding the effectiveness of "few-shot" prompting. First, from the collected data the "zero-shot" prompt usually produced many more trivial exercises, such as creating a list of several numbers, printing out a message with certain parameters, etc. These are not even at the "easy" level. This is understandable, as the "zero-shot" prompt does not have enough information about how an "easy" or "hard" exercise would be like. As a result, about 51.3% of "easy" exercises generated by "few-shot" prompting were ready-to-use, compared to just 32.9% of them in "zero-shot" prompting. This is similar to what was observed in a previous study (Ta et al., 2023), which used an older version of the GPT-3.5 Turbo model.

Second, we noted that "zero-shot" prompting could produce more duplicated exercises in response to each generation request. Note that each of ExGen's generation request requires 3 exercises to be generated. In quite a number of cases, 2 or even 3 out of the 3 generated exercises were exactly the same (in terms of problem statement, test cases, and solution). In our evaluation, for each response from the LLM we only counted the unique exercises which passed our manual inspection as ready-to-use. For example, if all 3 generated exercises passed the manual inspection and 2 of them were identical, we counted the number of ready-to-use exercises as 2 in this case.

The "few-shot" prompt produced much less duplicated exercises, as shown in Figure 9. We can see from the figure that for "zero-shot", the ratios between the numbers of ready-to-use exercises and the number of generated exercises (which could contain duplications)

Table 3 Performance (accuracy, cost, and time) of "zero-shot" vs. "few-shot" prompting when using GPT-3.5 Turbo with all keywords and exercise types. A total of 702 exercises were generated and evaluated for the data in this table

	zero-shot accuracy	zero-shot cost ($)	zero-shot time (s)	few-shot accuracy	few-shot cost ($)	few-shot time (s)
Easy	32.9%	0.0064	5.3	**51.3%**	0.0046	5.2
Intermediate	19.6%	0.0109	10.4	**20.9%**	0.0108	10
Hard	12.4%	0.0158	18.6	**13.7%**	0.0156	15.6

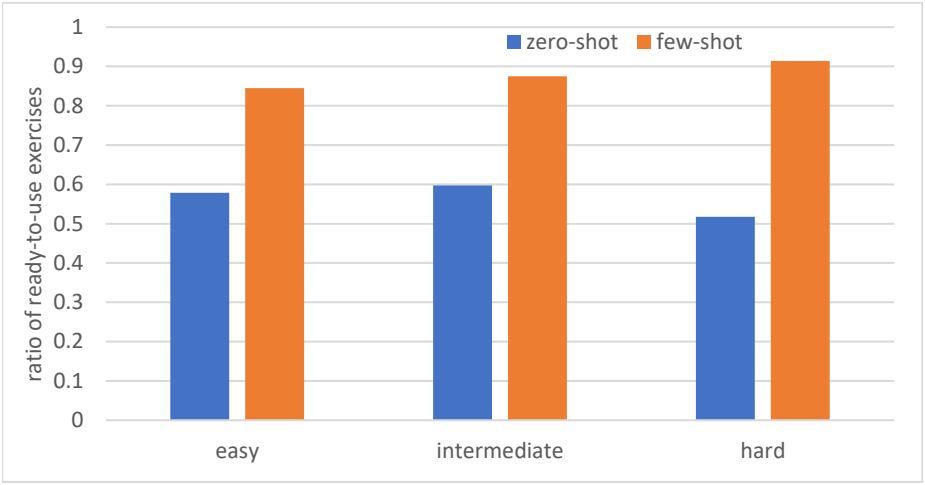

Fig. 9 Ratios of ready-to-use exercises vs. the total number of exercises passing manual inspection (higher is better). This is to show that "zero-shot" produced many duplicated exercises when responding to a generation request. Those exercises could pass manual inspection, but only one of them could be counted as ready-to-use.

which passed manual inspection are much lower than those for "few-shot". This shows that relevant examples given to the GPT-3.5 Turbo model as "shots" are particularly useful in providing more unique ready-to-use exercises. This translates to generally faster average generation times and less average costs, as shown in Table 3, even though "few-shot" prompting requires more tokens for the example exercises to be passed to the LLM.

Table 3 shows that generating ready-to-use "intermediate" and "hard" exercises is still quite challenging, even with recent advances in LLMs and latest updates to the OpenAI pretrained models. In the experiments, only around 20% of "intermediate" exercises generated by "zero-shot" and "few-shot" prompting were considered ready-to-use, respectively. Similarly, around 12-13% of generated "hard" exercises were ready-to-use. It was noted that "few-shot" prompting was better than "zero-shot" in this aspect, but not by much. We also noted that by providing more detailed information about difficulty levels with examples in the prompt, "few-shot" produced more exercises with the desired difficulty levels. However, quite a number of these exercises did not have clear problem statements, correct solutions and/or test cases, so they were not considered ready-to-use. Nonetheless, we noted that it's possible to make some additional manual modifications so that many of these more challenging exercises would become ready-to-use.

Table 3 also shows the average time and cost to produce a ready-to-use exercise using the two prompting strategies combined with the chained filters for various difficulty levels. We noted that the generation time was quite reasonable for "easy" exercises, which was around 5 seconds per exercise for both "few-shot" and "zero-shot" prompting. An "intermediate" exercise required around 10 seconds for both prompting approaches. The

"few-shot" strategy was faster than "zero-shot" on average (15.6 seconds vs. 18.6 seconds) for generating "hard" exercises. We noted that the results obtained here were significantly better compared to what had been reported previously (Ta et al., 2023), in which it took several minutes to produce a ready-to-use "hard" programming exercise. We attributed this improvement mainly to the latest update (Apr 2024) of the GPT-3.5 Turbo model which made it both more accurate and faster to run. Another reason is that our prompts define various difficulty levels more explicitly as compared to Ta et al. (2023). Finally, the set of keywords used in the prompts, especially the "complex" keywords could have positive impacts on "hard" exercise generation performance. Later in RQ-3 of this paper, we will present a more detailed analysis of these impacts.

The average cost (in $, Table 3) for generating an "easy" ready-to-use exercise is about 28% less when using "few-shot" prompting as compared to "zero-shot" prompting. The same table also shows that the costs for generating "intermediate" or "hard" ready-to-use exercise are slightly better (less expensive) with "few-shot" prompting. Although in our experiments "few-shot" generated more ready-to-use exercises, it also incurred more cost due to the examples required in each request as compared to "zero-shot". From Table 3, the average cost of generating ready-to-use exercises increases with the difficulty level required. This is because multiple candidate exercises will have to be generated before ExGen can find one that meets the "intermediate" or "hard" difficulty requirement and has correct solutions/test cases.

RQ1-Summary: The "few-shot" prompt outperformed "zero-shot" prompting in all cases in terms of accuracy, generation time and cost. This is partly because "few-shot", which incorporates relevant exercises as examples, produced more unique exercises at the correct level of difficulty. On average, the "few-shot" prompt was faster to run and incurred less cost compared to "zero-shot" prompting despite using more tokens due to the "shots". The latest update of the GPT-3.5 Turbo model (Apr 2024) brought significant improvements in average exercise generation time. However, the current accuracy level is still rather low for more challenging exercise generation. ExGen will need to generate many candidate exercises until it can obtain a ready-to-use one.

RQ2-Evaluating ExGen's auto-filtering approach

The standalone performance of the unit testing as well as the overall filtering performance (chain of all filters) are shown in Table 4 (higher percentage is better). From the experiments, we noted that the solutions of most generated exercises could pass the compilation check so we did not include its result in Table 4. In addition, to reduce the API cost, the difficulty check was only carried out if an exercise could pass the earlier filtering checks, i.e., compilation, and then unit testing, so Table 4 does not show the performance of the difficulty check separately.

Table 4 Performance of the auto-filtering checks (combined results for all keywords and example exercise types), GPT-3.5 Turbo. Difficulty check is not done if the generated exercises do not pass unit testing. The "few-shot" prompt produces better filtering performance for all types of exercises

	zero-shot (unit test)	zero-shot (chain)	few-shot (unit test)	few-shot (chain)
Easy	73.9%	73.5%	81.6%	**82.9%**
Intermediate	55.6%	71.8%	54.7%	**85.5%**
Hard	52.1%	62.4%	56.4%	**91.5%**

Table 4 shows that the unit testing check when used alone can correctly identify if a candidate exercise is ready-to-use or not in many cases, especially for "easy" exercises generated with "few-shot" prompting (81.6% accuracy when compared to the manual inspection's result). However, unit testing becomes less effective as the only filter for harder exercises as evidenced in Table 4. This is because although many candidates for harder exercises had working solutions and test cases in our experiments, they were not at the right levels of difficulty.

We observed good performance for our auto-filtering approach when chaining all the three filters, i.e., compilation followed by unit testing followed by difficulty check. The accuracy of the chained filters improved significantly for all prompting strategies when filtering "intermediate" and "hard" exercises. Overall, we noted that "few-shot" combined with the chained filters produced the best performance. Using this combination, we are more confident that ExGen can provide more ready-to-use exercises for students to practice on-demand.

It was also noted that the LLM used (GPT-3.5 Turbo) in ExGen tended to provide partially correct solutions and/or test cases when generating "intermediate" and "hard" exercises. Although these candidate exercises could meet the difficulty requirement, they failed ExGen's chained filters earlier, i.e., during unit testing, and were not checked for their difficulty level to save cost.

RQ2-Summary: The chained filtering mechanism worked better than individual filters, e.g., unit testing, in all cases. When combined with "few-shot" prompting, the chained filters provided the best filtering performance. Therefore, this combination is used as the default in ExGen for ready-to-use exercise generation.

RQ3-Impact of keyword types

In this section we report the impact of different keyword types on the accuracy, time, and cost of exercise generation. For brevity, we only report the results using "few-shot" prompting here, as it has been shown in RQ-1 and RQ-2 to be the better prompting strategy.

Figure 10 shows the accuracy, while Figures 11 and 12 show the average cost and time when using three different types of keywords, namely "simple", "complex", and "domain" (as listed in Table 2) for exercise generation. For each keyword type, we used all types of

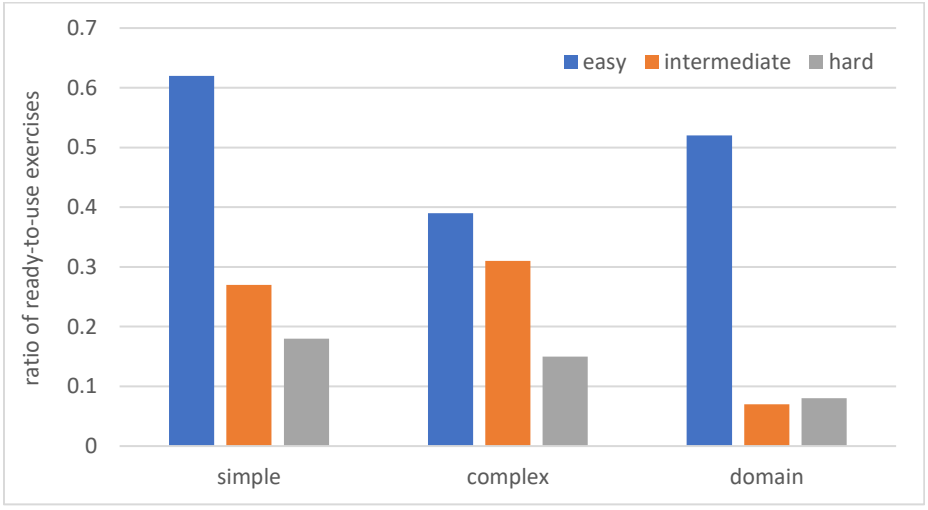

Fig. 10 The impact of different types of keywords on ready-to-use exercise generation's accuracy for "few-shot" prompting. The "simple" and "complex" keywords appear to perform much better than "domain" keywords for generating harder exercises.

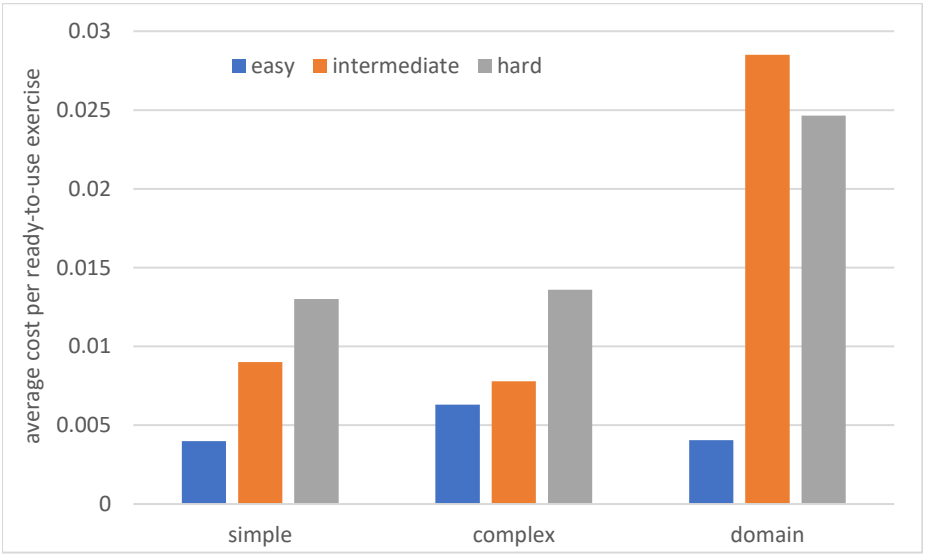

Fig. 11 The average cost (in $, less is better) per ready-to-use exercise when using different keyword types in "few-shot" prompting. The "domain" keywords have much higher cost compared to "simple" and "complex" when generating more challenging exercises.

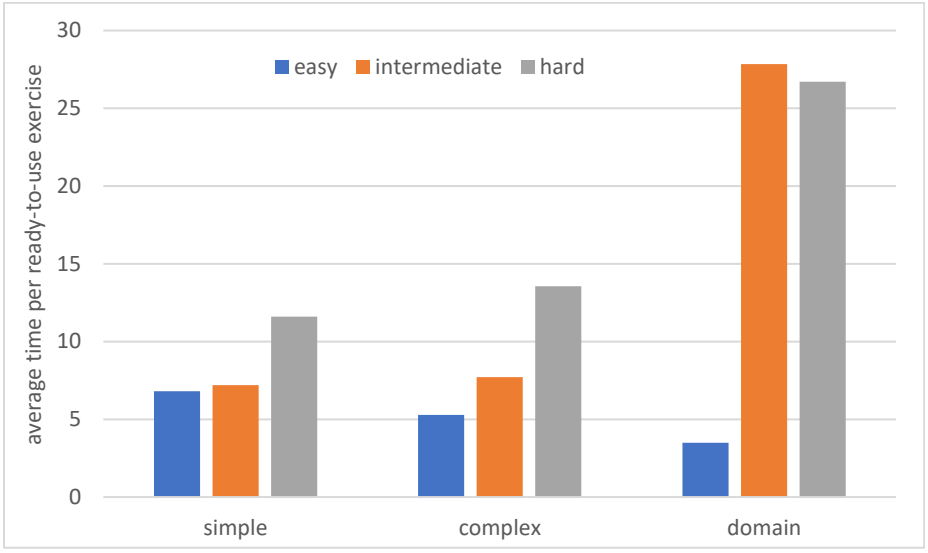

Fig. 12 The average time (in seconds) consumed per ready-to-use exercise when using different keyword types. Like the previous figures (10 and 11), using "domain" keywords requires much longer time to generate intermediate and hard exercises

problem ("concept" and "domain", as shown in Table 1) in the examples of the prompt. It is observed from Figure 10 that "simple" and "complex" keywords provide similarly decent accuracy levels when generating "intermediate" and "hard" exercises. However, Figure 10 also shows that "complex" keywords do not perform well for "easy" exercise generation. This is understandable, as "complex" keywords include more difficult programming concepts such as nested list, or multiple levels of nested loops, etc. It is challenging to create simple, easy exercises using these keywords. On the other hand, "simple" keywords appeared to be the most versatile, as they provided consistently high accuracy in exercise generation for all required difficulty levels in the experiments.

On the other hand, the "domain" type of keywords did not perform well when generating "intermediate" and "hard" exercises. Upon closer inspection, we noted that using "domain" keywords produced more interesting, novel, and practical exercises which could pass the unit testing filter. However, many of them are not difficult enough to be classified as "intermediate" or "hard". As a result, the average cost and time per each "intermediate" or "hard" ready-to-use exercise generated using "domain" keywords were much higher than those using the "simple" or "complex" keywords. This is shown in Figures 11 (average cost in $), and Figure 12 (average time in seconds). We concluded that "domain" keywords are most suitable for generating practical and novel "easy" level of programming exercises.

RQ3-Summary: The "simple" keyword type appeared to be the most versatile as ExGen could generate more ready-to-use exercises using it for all levels of difficulty. The "complex" keywords performed well for "intermediate" and "hard" exercise generation. The "domain" keywords were most suitable for "easy" exercise generation, but not for more challenging exercises.

RQ4-Impact of the types of example exercises

As shown in Figure 13, the "domain" type of programming exercises used as examples in "few-shot" prompting performs better in "easy" exercise generation when compared to "concept" exercises. However, the opposite is true when generating "intermediate" exercises. From what we observed when inspecting individual exercises generated, "domain" examples appeared to produce more new and practical exercises, but many of them were not difficult enough to be considered as "intermediate". There was not much difference in accuracy between using "domain" and "concept" examples to generate new "hard" exercises - not many of them were considered ready-to-use.

Figures 14 and 15 show the average cost and time per ready-to-use exercise generated using the two different example types. It can be seen in Figure 14 that "intermediate" exercise generation incurs more cost when using "domain" examples, compared to the case of using "concept" examples, due to the reason explained above. In the same figure, it is

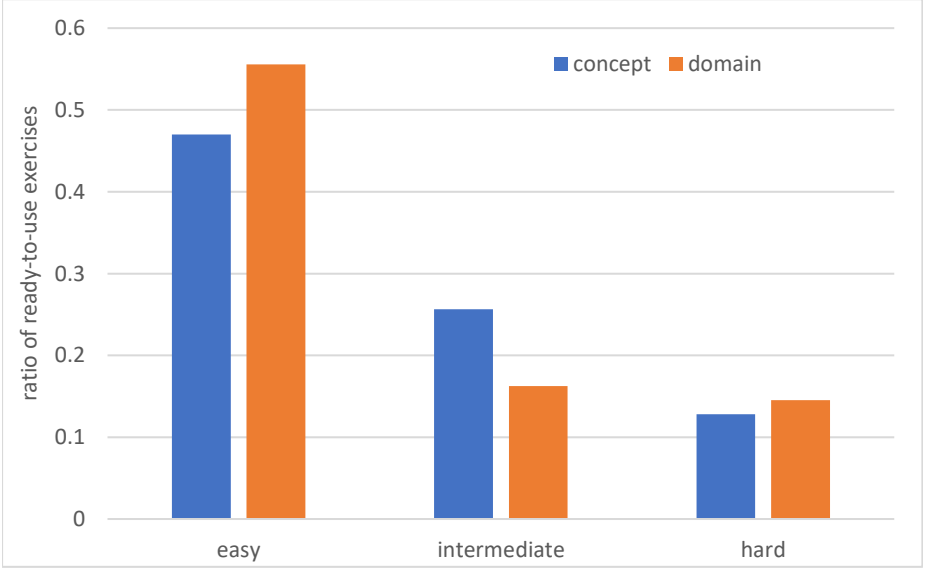

Fig. 13 The accuracy of "few-shot" prompting when generating exercises using two different example types. The "concept" examples seem to be better for "intermediate" exercises, while the "domain" examples have an edge for easy exercises

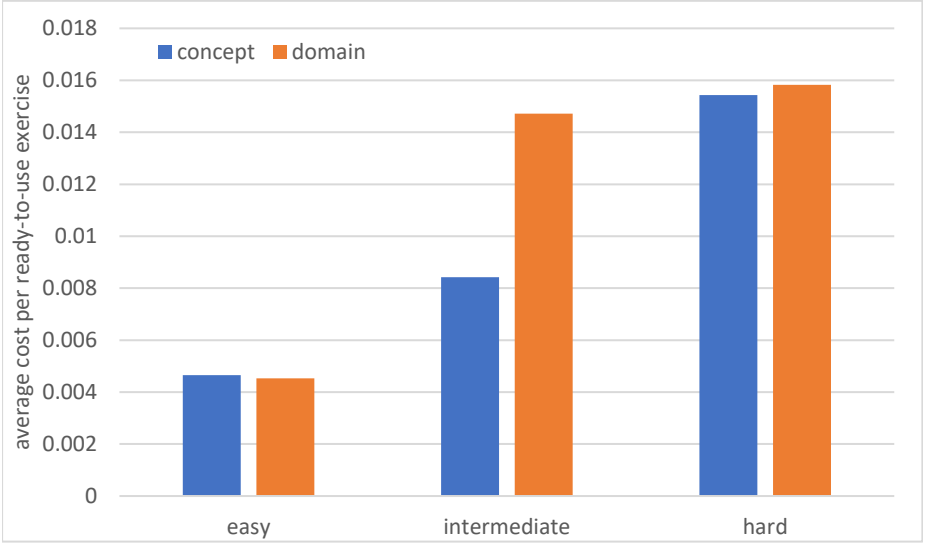

Fig. 14 The average cost per ready-to-use exercise for different types of examples used in "few-shot" prompting

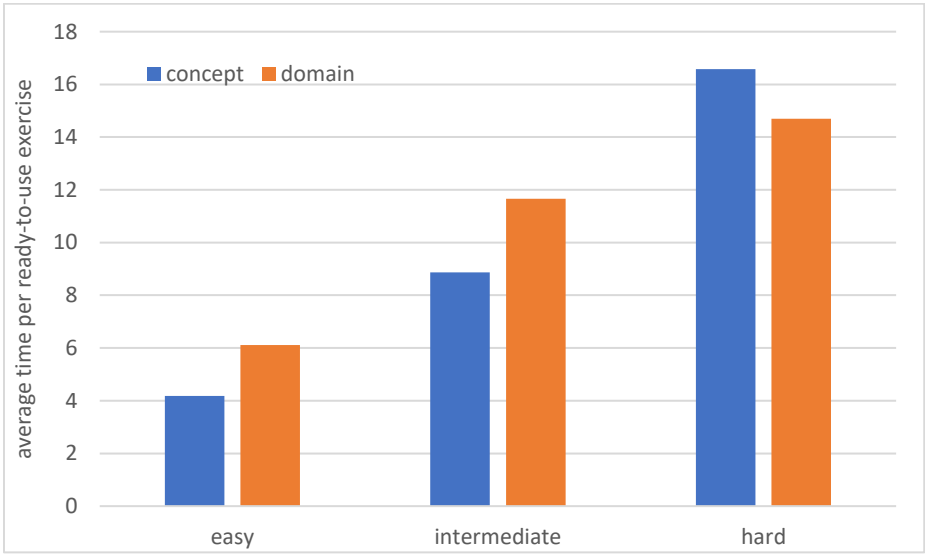

Fig. 15 The average time (in seconds) per ready-to-use exercise for different types of examples used in "few-shot" prompting

observed that "easy" and "hard" exercises have similar average costs. Finally, Figure 15 shows that the average times per ready-to-use exercise are not very different between different example types. When compared to the impact of keyword types (RQ-3), we noted that the impact of example types on accuracy as well as cost and generation time was not as significant.

RQ4-Summary: Different types of exercises used as examples in "few-shot" prompting appeared to have some impacts on the generation performance. In particular, "domain" examples were more accurate in "easy" exercise generation, while "concept" examples were better in "intermediate" exercise generation. However, example types did not seem to have a drastic impact on generation performance, as compared to keyword types.

RQ5-Impact of LLM models

In this section, we reported the performance of ExGen when the latest model as the time of experiments (GPT-4 Turbo, version gpt-4-0125-preview) was used versus the case when the much cheaper and faster GPT-3.5 Turbo (version gpt-3.5-turbo-0125) was used for exercise generation.

Table 5 shows a summary of performance measures obtained for GPT-4 Turbo and GPT-3.5 Turbo (extracted from Table 1, RQ-1). When compared to GPT-3.5 Turbo, we could see that GPT-4-Turbo performed much better in terms of accuracy when generating "intermediate" and "hard" exercises. Notably, with GPT-4 Turbo, the accuracy of "zero-shot" has improved significantly, especially in generating "intermediate" exercises. This can be explained by Figure 16, which shows the ratios of ready-to-use exercises versus exercises passing manual inspection (which might have duplications) with GPT-4 Turbo. From this figure, it is obvious that the gap between "zero-shot" and "few-shot" prompting is much smaller here compared to what was obtained using GPT-3.5 Turbo (shown in Figure 9, RQ-1). This means "zero-shot" using GPT-4 Turbo did not produce a lot of duplicated exercises, which translated into more ready-to-use exercises generated. However, as shown in Table 5, "few-shot" is still the better prompting strategy regardless of the LLM used.

Surprisingly, the newer GPT-4 Turbo model did not perform as well as the older model in "easy" exercise generation. We attributed this to the observation that GPT-4 Turbo tends to produce more challenging exercises with the keywords used in our experiments. As

Table 5 Performance (accuracy, cost, and time) of "zero-shot" vs. "few-shot" prompting using all keywords and example types. A total of 702 exercises were generated with GPT-4 Turbo and evaluated. The table also includes data obtained with GPT-3.5 Turbo (in parentheses) for easier comparison

	zero-shot accuracy	zero-shot cost ($)	zero-shot time (s)	few-shot accuracy	few-shot cost ($)	few-shot time (s)
Easy	36.8% (32.9%)	0.126 (0.0064)	15.1 (5.3)	45.3% (51.3%)	0.1134 (0.0046)	11.4 (5.2)
Intermediate	**36.3**% (19.6%)	0.13 (0.0109)	18.3 (10.4)	**37.2**% (20.9%)	0.139 (0.0108)	16.6 (10)
Hard	**17.9**% (12.4%)	0.223 (0.0158)	49.5 (18.6)	**29.1**% (13.7%)	0.1682 (0.0156)	23.4 (15.6)

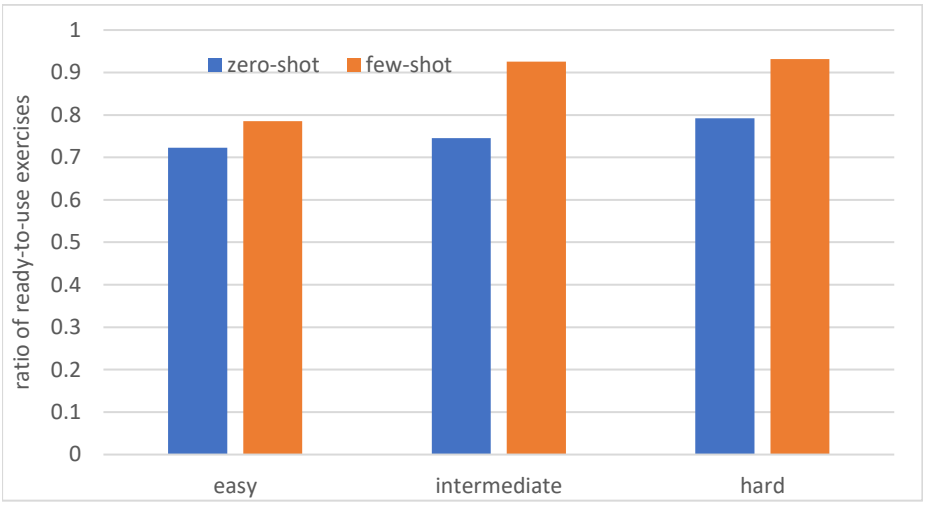

Fig. 16 Ratios of ready-to-use exercises vs. manually inspected exercises generated by "zero-shot" and "few-shot" prompting in GPT-4 Turbo

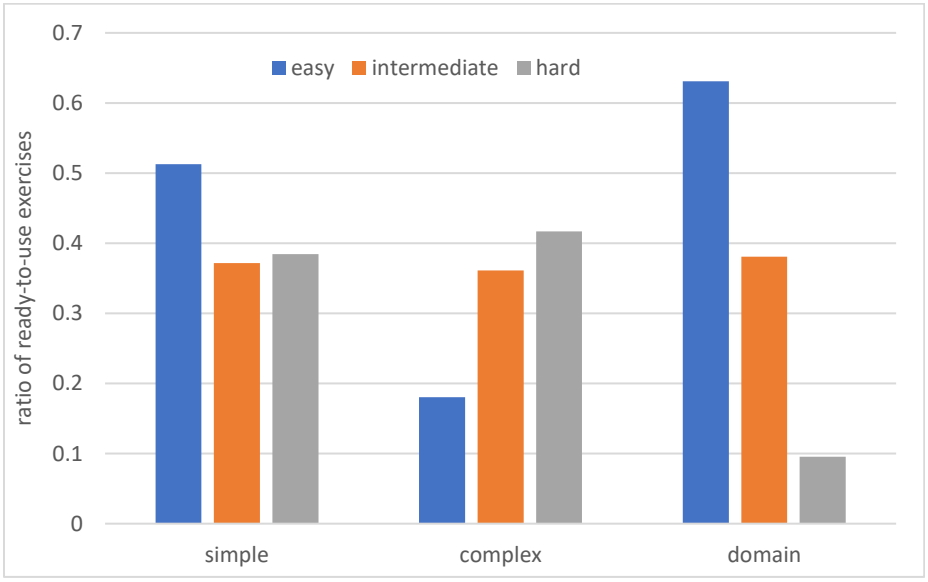

Fig. 17 The impact of keyword types in "few-shot" prompting with GPT-4 Turbo

shown in Figure 17, especially when the keywords were of "complex" type, GPT-4 Turbo produced many good programming exercises, but they were too difficult to be considered as "easy". The "simple" and "complex" keyword types worked best for "intermediate" and "hard" exercise generation. On the other hand, the "domain" keywords produced more ready-to-use "easy" exercises. When using different example types in "few-shot"

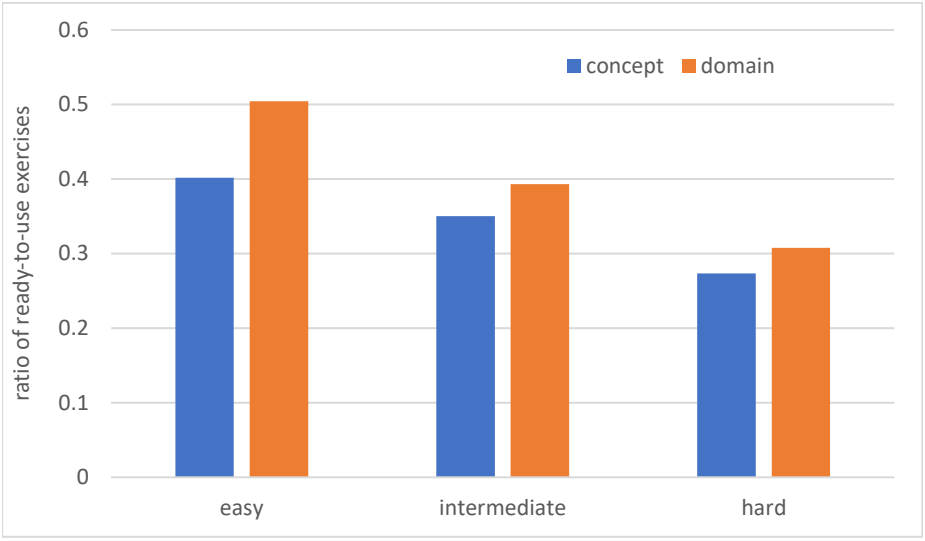

Fig. 18 The accuracy of "few-shot" prompting when generating exercises using two different example types, GPT-4 Turbo

prompting (Figures 18), we observed no significant impacts on generation performance for the more challenging exercises. This was in line with our previous observations using GPT-3.5 Turbo.

The newer GPT model is also more costly and much slower in terms of generation times; as shown in Table 5. This is not a surprise, as GPT-4 Turbo is believed to be a much larger model compared to GPT-3.5, although OpenAI did not reveal many details about the internal structure of the models. With more parameters, the quality of responses would be improved, i.e., the new model should make less mistakes and hallucinate less; but inference time might be slower. In addition, the cost of GPT-4 Turbo as the time of writing is 20x the cost of GPT-3.5 Turbo. We can conclude that for "intermediate" and "hard" exercise generation, if you need on-demand, ready-to-use exercises, and cost is not much of an issue, GPT-4 Turbo can be used. Otherwise, GPT-3.5 Turbo is still useful given its speed and the much cheaper price.

RQ5-Summary: GPT-4 Turbo provided much better accuracy when generating "intermediate" and "hard" exercises when compared to GPT-3.5 Turbo. It also had the tendency to generate more challenging exercises with the keywords used, so the accuracy for "easy" generation was not better than that of GPT-3.5 Turbo. GPT-4 Turbo also incurred much more cost and was significantly slower in our experiments. GPT-3.5 Turbo is still a good LLM for on-demand exercise generation due to its speed and lower pricing.

Threats to validity

We note that there are several limitations which may affect the validity of this study. First, the output from LLMs could be non-deterministic, which might impact the quality of generated exercises. We have tried to take this into account by generating and evaluating a large number of exercises (>1400). Second, manual evaluation of exercises by human experts could be subjective, especially when assigning a difficulty level and determining if the exercise is ready-to-use or not. To reduce this bias, we performed multiple rounds of manual assessment for each generated exercise. In each round, we compared generated exercises across different difficulty levels, and to existing exercises that have been manually crafted and used in our introductory programming course. Finally, as more advanced LLMs are being released, e.g., GPT-4o, it is possible that the results obtained from the new models are different from what we have seen in this study. We plan to continue our research into programming exercise generation with the latest models as they become available.

Related work

Much research has been done in automatic generation of formative feedback and reference solutions to code produced by students (Du et al., 2024; Keuning et al., 2018; Koutcheme, 2022; Ta et al., 2022; Tipirneni et al., 2024). Such feedback and reference solutions could be generated by pre-trained LLMs to help novice programmers know how to proceed when facing coding issues. On the other hand, not much has been done in generating ready-to-use programming exercises for students to do more practice (Ta et al., 2023). Jordan et al. (2024) investigated the usage of ChatGPT in generating programming exercises in native (non-English) languages. Logacheva et al. (2024) focused on contextually personalized programming exercises generated by generative AI. Kurdi et al. (2020) conducted a systematic review of automatic exercise generation in many different domains such as analytical reasoning, geometry, history, logic, relational databases, programming, and science.

Zavala and Mendoza (2018) used Automatic Item Generation (AIG) to address the problem of creating many similar programming exercises using pre-defined templates which are used for quizzes. The main goal was to ensure consistency in testing many students with questions of the same level of difficulty. On the other hand, ExGen focuses on generating ready-to-use exercises for a specific difficulty level and concept that the student is currently working on. We leveraged the latest advances in LLMs to autogenerate many novel exercises and filter them to ensure that they are suitable for students. Exercises considered in ExGen are different from other kinds of programming practices such as faded Parson problems (Fromont et al., 2023), which require students to fill in code in partially scrambled solutions.

The most relevant work in automatic exercise generation using pre-trained LLMs has been done by Sarsa et al. (2022). The authors explored OpenAI Codex (which has been deprecated since Mar 2023) for the purpose of creating new programming exercises and code explanations. They found that many Codex-generated exercises were sensible and novel, but may have confusing problem statements, missing or faulty test cases. In this work, we went further with the implementation and evaluation of an auto-filtering tool to supply students with ready-to-use exercises at various difficulty levels on-demand.

Most recently, Del Caprio Gutierrez et al. (2024) evaluated GPT-4 for contextualized programming exercise generation and found that the quality of generated problems (exercises) was high. However, the authors did not consider generating problems with different levels of difficulty. Our evaluation showed that generating challenging programming exercises is not a trivial task, even for the most advanced LLM. It was also not clear whether the problems generated and studied in Del Caprio Gutierrez et al. (2024) were ready-to-use or not. In Ta et al. (2023), the authors studied automatic programming exercise generation using GPT-3.5 Turbo. They did not consider the impact of various types of keywords and example exercises used as shots in the prompting strategies. They also did not consider the cost and the impact of the latest GPT models on the generation performance.

Recently, there has been an increase in the number of studies leveraging pre-trained LLMs for code understanding and educational purposes (Jury et al., 2024; Nam et al., 2024). Kasneci et al. (2023) discussed the potential benefits, for instance content generation and personalized learning, as well as challenges, e.g., model biases, system brittleness, etc., when applying LLMs to education. Similarly, Becker et al. (2023) elaborated the educational opportunities of AI code generation, and how educators should act quickly given these developments. Finnie-Ansley et al. (2023) reported the performance of OpenAI Codex on real questions from programming courses when compared to that of actual students. Denny et al. (2023) studied GitHub Copilot, a plug-in for IDEs like VS Code, which is based on the Codex model, to see what kinds of problems it would not perform well. The authors also found that prompt engineering can play an important role in interacting with AI tools like Copilot. MacNeil et al. (2023) found that code explanations such as line-by-line, high-level summary, and lists of important concepts generated by Codex and GPT-3 were helpful to most students. Kazemitabaar et al. (2023, 2024) studied how LLM's code generation capability could assist novice programmers via a controlled experiment for young students. We note that our work addresses a complementary and important aspect of programming education, which is automatic ready-to-use exercise generation, so that students would have more chance to practice and improve.

Conclusion

We have implemented and evaluated ExGen, a new software tool that automatically generates ready-to-use programming exercises for computing students. ExGen manages a personalized database of seed exercises and constructs appropriate LLM prompts to obtain candidate exercises for a requested difficulty level and user-specified keywords. The keywords might include learning concepts, or real-world, practical contexts. ExGen incorporates a novel combination of auto-filtering checks which are applied sequentially on the candidate exercises. The automated checks include software engineering techniques such as code compilation and unit testing, as well as LLM-based difficulty classification. The auto-filtering checks aim to reduce the tedious work of manually inspecting all the generated output and select ready-to-use exercises.

From the careful evaluation using more than 1400 auto-generated exercises, we believe that ExGen can provide students with on-demand, ready-to-use programming exercises customized at the right difficulty levels and user-supplied keywords. LLM prompting strategies having sufficient details such as examples (shots) have been shown to be more accurate, i.e., to generate more exercises that are ready-to-use. In most cases, our auto-filtering approach with a chain of software engineering and LLM-based checks could work well regardless of the prompting strategy used. Furthermore, our evaluation provided insights into the impacts of keywords and example exercise types, as well as more advanced and expensive LLMs on generation performance, i.e., accuracy, cost, and time. Although newer models such as GPT-4 Turbo generally provide better accuracy, it is noted that automatically generating truly ready-to-use, "hard" programming exercises is still a problem which needs further research despite recent progress in LLM and AI development.

Abbreviations

AIG: Automatic Item Generation; GPT: Generative Pre-training Transformer; LLMs: Large Language Models; RQ: Research Question; VS: Visual Studio.

Acknowledgements

This research is supported by the Ministry of Education, Singapore, under its Tertiary Education Research Fund (Award No. MOE2021-TRF-014). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of the Ministry of Education, Singapore.

Authors' contributions

Nguyen Binh Duong Ta: carried out most aspects of this work include problem formulation, design, implementation, evaluation, and paper writing. Hua Gia Phuc Nguyen: conducted implementation and experiments. Swapna Gottipati: reviewed and improved the paper.

Authors' information

Nguyen Binh Duong Ta is an SMU Associate Professor in Computer Science (Education), School of Computing and Information Systems, Singapore Management University. He obtained his PhD in computer science from Nanyang Technological University, Singapore, 2006. His main interests are AI in education and distributed computing.

Hua Gia Phuc Nguyen was a Research Engineer at School of Computing and Information Systems, Singapore Management University.

Swapna Gottipati is an SMU Associate Professor of Information Systems (Education), and Associate Dean (Undergraduate Education), School of Computing and Information Systems, Singapore Management University. She obtained her PhD in information systems from Singapore Management University, Singapore, 2014. Her main interests are in educational technologies and digital business.

Funding
This research is supported by the Ministry of Education, Singapore, under its Tertiary Education Research Fund (Award No. MOE2021-TRF-014). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of the Ministry of Education, Singapore.

Availability of data and materials
The data might be provided upon request, subject to permission from the Ministry of Education, Singapore.

Declarations

Competing interests
The authors declare that they have no competing interests.

Author details
Nguyen Binh Duong Ta, donta@smu.edu.sg. Hua Gia Phuc Nguyen, hgpnguyen@smu.edu.sg. Swapna Gottipati, swapnag@smu.edu.sg. Affiliation: School of Computing and Information Systems, Singapore Management University, 80 Stamford Rd, Singapore 178902.

Received: 31 May 2024 Accepted: 23 April 2025
Published online: 1 January 2026 (Online First: 2 September 2025)

References

Arawjo, I., Swoopes, C., Vaithilingam, P., Wattenberg, M., & Glassman, E. L. (2024, May). ChainForge: A visual toolkit for prompt engineering and LLM hypothesis testing. In F. F. Mueller, P. Kyburz, J. R. Williamson, C. Sas, M. L. Wilson, P. Toups Dugas & I. Shklovski (Eds.), *Proceedings of the CHI Conference on Human Factors in Computing Systems* (pp. 1–18). ACM. https://doi.org/10.1145/3613904.3642016

Becker, B. A., Denny, P., Finnie-Ansley, J., Luxton-Reilly, A., Prather, J., & Santos, E. A. (2023). Programming is hard-or at least it used to be: Educational opportunities and challenges of AI code generation. In M. Doyle, B. Stephenson, B. Dorn, L.-K. Soh & L. Battestilli (Eds.), *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (pp. 500–506). ACM. https://doi.org/10.1145/3545945.3569759

Del Carpio Gutierrez, A., Denny, P., & Luxton-Reilly, A. (2024, March). Evaluating automatically generated contextualised programming exercises. In B. Stephenson, J. A. Stone, L. Battestilli, S. A. Rebelsky & L. Shoop (Eds.), *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1* (pp. 289–295). ACM. https://doi.org/10.1145/3626252.3630863

Denny, P., Kumar, V., & Giacaman, N. (2023). Conversing with Copilot: Exploring prompt engineering for solving CS1 problems using natural language. In M. Doyle, B. Stephenson, B. Dorn, L.-K. Soh & L. Battestilli (Eds.), *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (pp. 1136–1142). ACM. https://doi.org/10.1145/3545945.3569823

Du, X., Liu, M., Wang, K., Wang, H., Liu, J., Chen, Y., Feng, J., Sha, C., Peng, X., & Lou, Y. (2024, April). Evaluating large language models in class-level code generation. In A. Paiva, R. Abreu, A. Roychoudhury & M. Storey (Eds.), *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (pp. 1–13). ACM. https://doi.org/10.1145/3597503.3639219

Finnie-Ansley, J., Denny, P., Luxton-Reilly, A., Santos, E. A., Prather, J., & Becker, B. A. (2023, January). 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* (pp. 97–104). ACM. https://doi.org/10.1145/3576123.3576134

Fromont, F., Jayamanne, H., & Denny, P. (2023). Exploring the difficulty of Faded Parsons problems for programming education. In *Proceedings of the 25th Australasian Computing Education Conference* (pp. 113–122). ACM. https://doi.org/10.1145/3576123.3576136

Jordan, M., Ly, K., & Soosai Raj, A. G. (2024, March). Need a programming exercise generated in your native language? ChatGPT's got your back: Automatic generation of non-English programming exercises using OpenAI GPT-3.5. In B. Stephenson, J. A. Stone, L. Battestilli, S. A. Rebelsky & L. Shoop (Eds.), *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1* (pp. 618–624). ACM. https://doi.org/10.1145/3626252.3630897

Joshi, I., Budhiraja, R., Dev, H., Kadia, J., Ataullah, M. O., Mitra, S., Kumar, D., & Akolekar, H. D. (2023). *ChatGPT--A blessing or a curse for undergraduate computer science students and instructors?* arXiv preprint arXiv:2304.14993.

Jury, B., Lorusso, A., Leinonen, J., Denny, P., & Luxton-Reilly, A. (2024, January). Evaluating LLM-generated worked examples in an introductory programming course. In N. Herbert & C. Seton (Eds.), *Proceedings of the 26th Australasian Computing Education Conference* (pp. 77–86). ACM. https://doi.org/10.1145/3636243.3636252

Kasneci, E., Sessler, K., Küchemann, S., Bannert, M., Dementieva, D., Fischer, F., Gasser, U., Groh, G., Günnemann, S., Hüllermeier, E., Krusche, S., Kutyniok, G., Michaeli, T., Nerdel, C., Pfeffer, J., Poquet, O., Sailer, M., Schmidt, A., Seidel, T., … & Kasneci, G. (2023). ChatGPT for good? On opportunities and challenges of large language models for education. *Learning and Individual Differences*, *103*, 102274. https://doi.org/10.1016/j.lindif.2023.102274

Kazemitabaar, M., Chow, J., Ma, C. K. T., Ericson, B. J., Weintrop, D., & Grossman, T. (2023). Studying the effect of AI code generators on supporting novice learners in introductory programming. In A. Schmidt, K. Väänänen, T. Goyal, P. O. Kristensson, A. Peters, S. Mueller, J. R. Williamson & M. L. Wilson (Eds.), *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (pp. 1–23). ACM. https://doi.org/10.1145/3544548.3580919

Kazemitabaar, M., Ye, R., Wang, X., Henley, A. Z., Denny, P., Craig, M., & Grossman, T. (2024, May). Codeaid: Evaluating a classroom deployment of an LLM-based programming assistant that balances student and educator needs. In F. F. Mueller, P. Kyburz, J. R. Williamson, C. Sas, M. L. Wilson, P. Toups Dugas & I. Shklovski (Eds.), *Proceedings of the CHI Conference on Human Factors in Computing Systems* (pp. 1–20). ACM. https://doi.org/10.1145/3613904.3642773

Keuning, H., Jeuring, J., & Heeren, B. (2018). A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE)*, *19*(1), 1–43. https://doi.org/10.1145/3231711

Koutcheme, C. (2022). Towards open natural language feedback generation for novice programmers using large language models. In I. Jormanainen & A. Petersen (Eds.), *Proceedings of the 22nd Koli Calling International Conference on Computing Education Research* (pp. 1–2). ACM. https://doi.org/10.1145/3564721.3565955

Koutcheme, C., Dainese, N., Sarsa, S., Hellas, A., Leinonen, J., & Denny, P. (2024). Open source language models can provide feedback: Evaluating LLMs' ability to help students using GPT-4-as-a-judge. In M. Monga, V. Lonati, E. Barendsen, J. Sheard & J. Paterson (Eds.), *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (pp. 52–58). ACM. https://doi.org/10.1145/3649217.3653612

Kurdi, G., Leo, J., Parsia, B., Sattler, U., & Al-Emari, S. (2020). A systematic review of automatic question generation for educational purposes. *International Journal of Artificial Intelligence in Education*, *30*, 121–204. https://doi.org/10.1007/s40593-019-00186-y

Liu, J., Xia, C. S., Wang, Y., & Zhang, L. (2024). Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, *36*.

Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., & Neubig, G. (2023). Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, *55*(9), 1–35. https://doi.org/10.1145/3560815

Logacheva, E., Hellas, A., Prather, J., Sarsa, S., & Leinonen, J. (2024, August). Evaluating contextually personalized programming exercises created with generative AI. In P. Denny, L. Porter, M. Hamilton & B. Morrison (Eds.), *Proceedings of the 2024 ACM Conference on International Computing Education Research-Volume 1* (pp. 95–113). ACM. https://doi.org/10.1145/3632620.3671103

MacNeil, S., Tran, A., Hellas, A., Kim, J., Sarsa, S., Denny, P., Bernstein, S., & Leinonen, J. (2023). Experiences from using code explanations generated by large language models in a web software development e-book. In M. Doyle, B. Stephenson, B. Dorn, L.-K. Soh & L. Battestilli (Eds.), *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (pp. 931–937). ACM. https://doi.org/10.1145/3545945.3569785

Nam, D., Macvean, A., Hellendoorn, V., Vasilescu, B., & Myers, B. (2024, April). Using an LLM to help with code understanding. In A. Paiva, R. Abreu, A. Roychoudhury & M. Storey (Eds.), *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (pp. 1–13). ACM. https://doi.org/10.1145/3597503.3639187

Sarsa, S., Denny, P., Hellas, A., & Leinonen, J. (2022). Automatic generation of programming exercises and code explanations using large language models. In J. Vahrenhold, K. Fisler, M. Hauswirth & D. Franklin (Eds.), *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1* (pp. 27–43). ACM. https://doi.org/10.1145/3501385.3543957

Ta, D., Nguyen, P., & Gottipati, S. (2023). ExGen: Ready-to-use exercise generation in introductory programming courses. In J. L. Shih et al. (Eds.), *Proceedings of the 31st International Conference on Computers in Education* (pp. 104–113). Asia-Pacific Society for Computers in Education.

Ta, D., Shar, l. K., & Shankararaman, V. (2022). AP-coach: Formative feedback generation for learning introductory programming concepts. In *Proceedings of IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)* (pp. 323–330). IEEE. https://doi.org/10.1109/TALE54877.2022.00060

Tipirneni, S., Zhu, M., & Reddy, C. K. (2024). Structcoder: Structure-aware transformer for code generation. *ACM Transactions on Knowledge Discovery from Data*, *18*(3), 1–20. https://doi.org/10.1145/3636430

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Liu, J., Awadallah, A. H., White, R. W., Burger, D., & Wang, C. (2023). *AutoGen: Enabling next-gen LLM applications via multi-agent conversation framework*. arXiv preprint arXiv:2308.08155.

Zavala, L., & Mendoza, B. (2018). On the use of semantic-based AIG to automatically generate programming exercises. In T. Barnes, D. Garcia, E. K. Hawthorne & M. A. Pérez-Quiñones (Eds.), *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (pp. 14–19). ACM. https://doi.org/10.1145/3159450.3159608

Publisher's Note