2023 Congress in Computer Science, Computer Engineering, & Applied Computing (CSCE)

Digital Rubber Duck: Leveraging Large Language Models for Extreme Programming

Timothy Elvira, Tyler Thomas Procko, Juan Ortiz Couder, Omar Ochoa
Department of Electrical Engineering and Computer Science
Embry-Riddle Aeronautical University
Daytona Beach, Florida, United States of America
{elvirat, prockot, ortizcoj}@my.erau.edu, ochoao@erau.edu

2023 Congress in Computer Science, Computer Engineering, &amp; Applied Computing (CSCE) | 979-8-3503-2759-5/23/$31.00 ©2023 IEEE | DOI: 10.1109/CSCE60160.2023.00051

Abstract—**The recent prevalence of Large Language models (LLMs), e.g., GPT-3.5 and GPT-4, has brought about a new age of man-computer symbiosis, where LLMs are employed for a litany of creative, constructive, scientific, or otherwise content-generative tasks, e.g., as general chatbot assistants, writing editors, digital subject matter experts, programming consultants, and so on. Of interest to software engineers is the concept of "rubber duck debugging", which is the act of expressing code, line-by-line, in natural language, to an inanimate object, e.g., a rubber duck, for the purpose of elucidating potential issues that can then be corrected. In this paper, we detail a workflow process that leverages the concept of rubber duck debugging, replacing the duck with a capable LLM, e.g., GPT-4. We call it Digital Rubber Duck Programming. Furthermore, the Extreme Programming (XP) method, an implementation of the Agile paradigm, is considered as easily integrated with the proposed workflow, as XP is performed in pairs (much like the modern software engineer works in pairwise fashion with an LLM) and because XP places emphasis on performing extensive code reviews and unit testing all code, which capable LLMs like GPT-4 can facilitate.**

Keywords—Software Engineering, Code Refactoring, Walkthroughs, Extreme Programmaing, GPT-4, ChatGPT

Paper Type – "Regular research Paper"

I. INTRODUCTION

ChatGPT is a revolutionary Large Language Model (LLM) containing billions of parameters [1, 2, 3]. OpenAI's state-of-the-art model can few-shot and even zero-shot learn to understand and generate from a user's query [4]. Few-shot and zero-shot learning is the ability to generalize learned information into a new context. This new domain may or may not be in the original training dataset [3]. The few- and zero-shot learning ability of ChatGPT allows the model to be used in a variety of domains. The model's domain agnosticism, paired with its advanced natural language understanding and generation, establishes ChatGPT as a state-of-the-art technology. Moreover, ChatGPT differs from other LLMs due to its training set also containing source code, empowering the model to generate code snippets in a plethora of coding languages. The unique combination of code and natural language situates ChatGPT as a possible Software Engineering (SWE) tool for computer scientists and software engineers. Many software engineers utilize the Agile lifecycle model to produce production-quality software within industry [5]. One such method is Extreme Programming (XP), or the production of software in pairs of engineers [6]. This paper proposes that ChatGPT, with its powerful code and natural language understanding, can act as a virtual, hyper-intelligent, ever-present programming partner to a software engineer using XP. This paper examines scenarios in which GPT is used in a few XP activities like code refactoring, code walkthroughs, and coverage testing.

The organization of this paper is as follows. Section II outlines the background knowledge required to understand the content related to XP and LLMs. Section III presents activities, prompts, and results from various XP activities. Section IV presents an overview of similar work that focuses on leveraging GPT for SWE or related activities. Finally, section V contains the authors' suggested future work and concluding statements.

II. BACKGROUND

A. Software Engineering

Software engineering as a discipline has been extant since at least the late 1950s, with discussions on software process models (e.g., Waterfall) occurring since the 1970s [7, 8, 9]. Under the traditional lifecycle model, software development requires a period of deliberation and documentation before any development begins. This traditional method of development was process-oriented, with little emphasis given to teams and their formation [10]. By its nature of up-front deliberation, the traditional development life cycle is extremely inflexible regarding change, e.g., modified customer requirements.

The Agile approach aimed to mitigate the rigidity of Waterfall by being customer-centric and adaptable to change. The Agile approach was conceived by a small group of expert software practitioners, who prefer "individuals and interactions over processes and tools;

979-8-3503-2759-5/23/$31.00 ©2023 IEEE
DOI 10.1109/CSCE60160.2023.00051

Authorized licensed use limited to: Hebei University of Science and Technology. Downloaded on May 11,2024 at 06:54:02 UTC from IEEE Xplore. Restrictions apply.

working software over comprehensive documentation; customer collaboration over contract negotiation; and responding to change over following a plan" [11]. Because of the lack of life cycle structure in Agile, there is a higher need for effective working teams. Extreme Programming (XP) is the most widely used Agile method, sharing the values espoused by the Agile Manifesto for Software Development. XP goes further to specify a simple set of practices, most notably, pair programming, in which two people write code together at the same workstation [6]. Pair programming with XP has been shown to improve the grades of student programmers, and the quality of student code, because of the benefits of actively walking through written code with another individual [12].

The three main activities covered in this paper are walkthroughs, method extraction & code refactoring, and code coverage & testing.

Walkthroughs:

Walkthroughs, a type of code review, are an SWE activity where engineers review and step through a code snippet to ensure it is correct and meets specifications [13]. During walkthroughs, reviewers will examine the code to find deviations from these specifications. Code walkthroughs have evolved over the years, including tools to assist reviewers in finding code errors, and identifying invalid functionality against specifications [14]. Originally, in the 1960s, walkthroughs were performed by someone paraphrasing and presenting logical algorithms aloud to others [15]. The intended purpose was not to find errors, but to train newcomers and introduce them to the system. Overtime, walkthroughs became useful in finding code defects, and so the goal changed from a training mechanism to a quality assurance activity. Currently, walkthroughs have adapted to fast-paced life cycles like Agile by following a lightweight framework [14]. This framework can be summarized in five key steps: creating, previewing, commenting, addressing feedback, and approving [14].

Code Review Flow	Action
Creating	Author modifies code
Previewing	Author visualizes code changes
Commenting	Reviewers comment on changes
Addressing Feedback	Author address comments
Approving	Reviewers accept or reject authors changes w.r.t comments.

Table 1. Steps to a code walkthrough

Method Extraction and Code refactoring:

Method extraction is a well-known and useful technique in wrangling large classes or methods [16]. Method extraction is a form of code refactoring that aims at making code cleaner and more readable by taking parts of code and refactoring them into a method. Method extraction is particularly difficult due to the need to preserve semantics and code behavior [17]. Typically, XP builds on the practice of refactoring by adding the benefit of a peer to ensure refactoring is done sensibly and in compliance with the specifications [18].

Method extraction falls under general code refactoring. General code refactoring operates on changing code to avoid poor coding designs, called code-smells [19]. Ideally, the goal of refactoring is to ensure a high-quality product is delivered to the customer. Quality is increased when code is highly cohesive and rarely coupled. Cohesion is code, either methods or classes, working together while coupling is code that is tightly bound together [19]. Because XP enforces multi-perspective and continual design, code refactoring is a natural activity in XP [20]. A peer ensures that the correct refactoring method is addressing a code smell in XP.

Code Coverage & Testing:

Code coverage is a software testing metric that determines the number of lines of code that are validated by a test procedure, which helps to analyze how comprehensive some software is verified [21].

B. Machine Learning

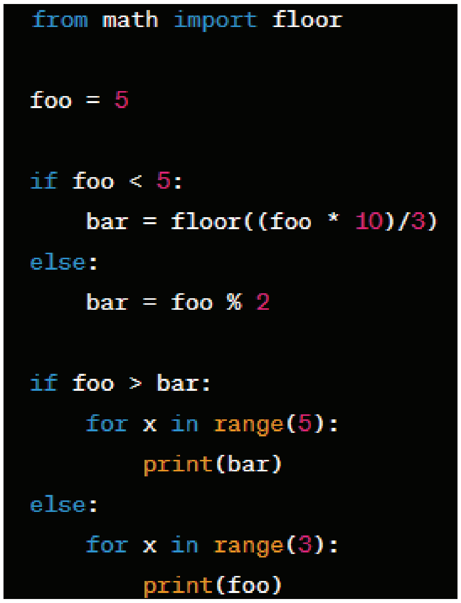
With the advent of Machine Learning (ML), various software engineering activities have been integrated with ML systems to assist the software developer. For instance, some of the earliest uses of ML in the Software Development Life Cycle (SDLC) include estimating development effort, classifying fault-prone software modules, modularizing source code and generating software test data [22].

The advancement of neural networks and particular activation functions, e.g., ReLU [23], in combination with expandingly powerful graphics processing units (GPUs) in computers, allowed neural networks (which, in the 1990s, were relatively shallow) to become deeper, i.e., Deep Learning, incorporating more layers and complex architectures [24, 25, 26].

In 2017, the groundbreaking neural network architecture, the Transformer, which uses a novel self-attention algorithm, was proposed [27]. Based upon the Transformer, contemporary, pre-trained LLMs with billions of parameters, like OpenAI's GPT series [1, 2, 3], Google's Fine-tuned Language Net (FLAN) [28] and Bidirectional Encoder Representations from Transformers (BERT) [29] models, and Facebook's Bidirectional and Auto-Regressive Transformers (BART) [30] and Large Language Model Meta AI (LLaMA) [31] models are used to power applications, provide insights to inquirers and assist in a variety of tasks.

Based on transformers, the code generation tool Intellicode Compose is an add-on for Integrated Development Environments (IDEs), e.g., Visual Studio Code, used by developers as a programming assistant [32]. Released in 2021 by OpenAI, Codex powers

Authorized licensed use limited to: Hebei University of Science and Technology. Downloaded on May 11,2024 at 06:54:02 UTC from IEEE Xplore. Restrictions apply.



```
from math import floor

foo = 5

if foo < 5:
    bar = floor((foo * 10)/3)
else:
    bar = foo % 2

if foo > bar:
    for x in range(5):
        print(bar)
else:
    for x in range(3):
        print(foo)
```

Fig. 1. The sample code snippet used in each SWE activity.

GitHub's Copilot. GitHub Copilit is described as an "AI pair programmer", that can integrate with certain IDEs, providing suggestions while coding [33]. GitHub Copilot has been evaluated as being useful for simpler software development tasks, where the human developer is needed finalize its output [34, 35]. While GitHub Copilot is a useful tool for software developers, the quality of its generated code depends heavily on the conciseness and correctness of the natural language prompts provided by the developer [36]. In 2022, OpenAI deprecated the Codex model, because GPT-3 and GPT-3.5 (ChatGPT) performed exceptionally well at coding tasks, as well as various other tasks.

Since mid-2022, LLMs for writing code have been discussed and shown to be quite useful for certain programming activities, e.g., autocompleting redundant portions of code, refactoring code, writing simple functions, etc. [37]. In the early months of 2023, LLM development accelerated rapidly, with ChatGPT (GPT-3.5 and GPT-4) becoming extremely capable programming assistants available to the mass public through a simple web interface. Building on GPT-3.5, GPT-4 has advanced reasoning capabilities and increased attention, in addition to having the distinction as being the first multimodal model in mass use, as it can accept image and text inputs, responding in text [38]. GPT has even recently been used to generate programming exercises and explanations for teaching students code [39].

III. DIGITAL XP ACTIVITIES

Our approach is to use ChatGPT in XP activities, taking note of input queries and ChatGPT's output. To illustrate this approach, consider the following simple snippet as an example of an engineer developing a piece of software alongside ChatGPT. This code snippet is in the Python language and uses variables, conditionals, and loops. The goal and meaning of the code are arbitrary. Fig. 1 is the code snippet used in all the scenarios. The following sections use this code sample for ChatGPT's application in the XP activities. This paper presents an approach to developing workflows that incorporates ChatGPT into software activities.

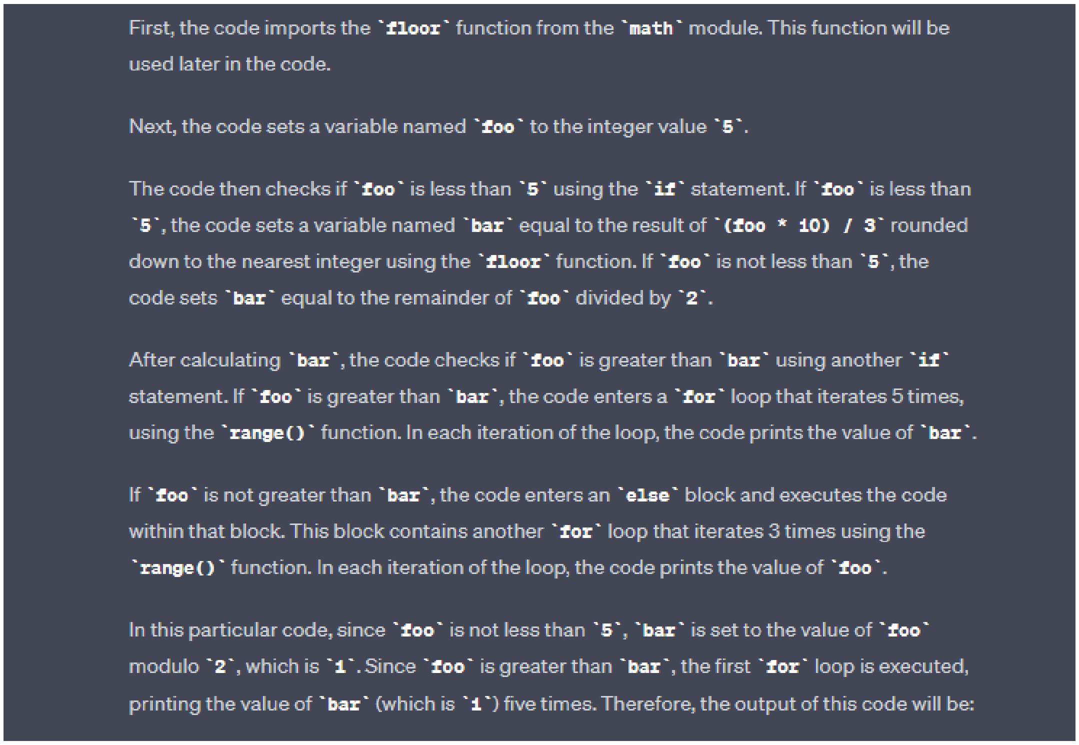
A. Walkthroughs

ChatGPT can conduct code walkthroughs to some extent. The query for walkthroughs is a simple statement of "Can you perform a walkthrough of this code" with the code snippet attached. As seen in Fig. 2, ChatGPT steps through and reads the snippet, shown in Fig. 1, line-by-line. For each line, ChatGPT provides a small summarization of code and the logical progression of variables. The generated transcript reads similarly to the original code walkthrough process, where an engineer would read aloud their logic. The last paragraph of the generated explanation is critical as ChatGPT hypothetically runs the code to find the behavior of the snippet which is printing '1' five times. ChatGPT's code walkthrough operates closely to the original walkthrough process rather than the steps found in Sadowski et al.'s work. This is not to say that ChatGPT cannot adhere to those five walkthrough steps but when simply prompting about a code walkthrough, ChatGPT resorts to just logically stepping through code. Adjusting the query by providing further walkthrough steps might enforce the model to generate a response that adheres to the five walkthrough steps. Example prompts to adhere GPT to the five walkthrough steps can be found in Table 2:

Code review flow	Prompt
Creating	- Refactor method X - Method extract lines 34-39
Previewing	- From these two code snippets, detail the difference between them
Commenting	- From these two code snippets, comment on the different [coding styles, execution, output, behavioral changes] - Do the code changes meet these given requirements?
Addressing Feedback	- Can you address if the additional changes meet given requirements
Approving	- Is this code snippet fulfilled given X criteria? - Is this code acceptable for production?

Table 2: Prompts to use for adhering ChatGPT to follow a structured walkthrough.

Authorized licensed use limited to: Hebei University of Science and Technology. Downloaded on May 11,2024 at 06:54:02 UTC from IEEE Xplore. Restrictions apply.



First, the code imports the `floor` function from the `math` module. This function will be used later in the code.

Next, the code sets a variable named `foo` to the integer value `5`.

The code then checks if `foo` is less than `5` using the `if` statement. If `foo` is less than `5`, the code sets a variable named `bar` equal to the result of `(foo * 10) / 3` rounded down to the nearest integer using the `floor` function. If `foo` is not less than `5`, the code sets `bar` equal to the remainder of `foo` divided by `2`.

After calculating `bar`, the code checks if `foo` is greater than `bar` using another `if` statement. If `foo` is greater than `bar`, the code enters a `for` loop that iterates 5 times, using the `range()` function. In each iteration of the loop, the code prints the value of `bar`.

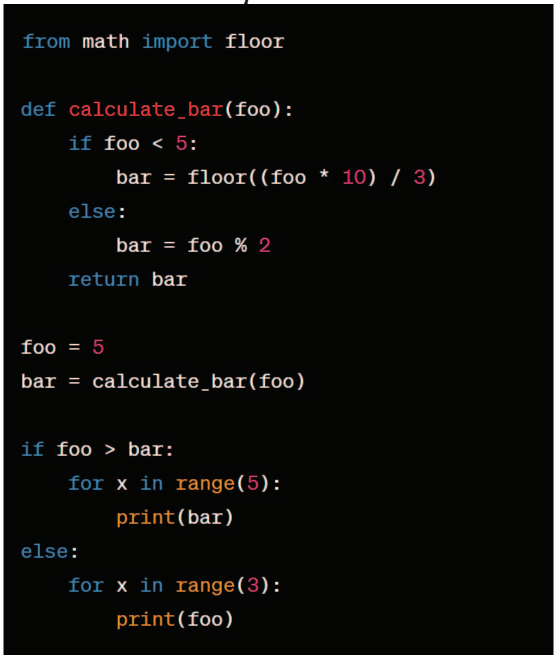
If `foo` is not greater than `bar`, the code enters an `else` block and executes the code within that block. This block contains another `for` loop that iterates 3 times using the `range()` function. In each iteration of the loop, the code prints the value of `foo`.

In this particular code, since `foo` is not less than `5`, `bar` is set to the value of `foo` modulo `2`, which is `1`. Since `foo` is greater than `bar`, the first `for` loop is executed, printing the value of `bar` (which is `1`) five times. Therefore, the output of this code will be:

Fig. 2. ChatGPT's response to "Can you perform a walkthrough of this code?"

More prompt testing is required to learn the full extent of ChatGPT's capabilities in conducting code walkthroughs; however, it currently produces a modest and useful walkthrough as a peer in XP without prompting for specific structured walkthroughs.

Prompt: ***Can you extract a method for the first if/else pair?***



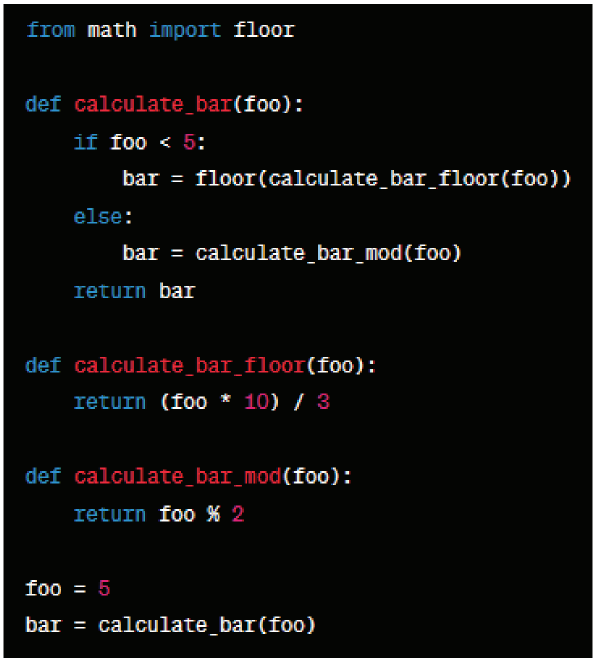
```
from math import floor

def calculate_bar(foo):
    if foo < 5:
        bar = floor((foo * 10) / 3)
    else:
        bar = foo % 2
    return bar

foo = 5
bar = calculate_bar(foo)

if foo > bar:
    for x in range(5):
        print(bar)
else:
    for x in range(3):
        print(foo)
```

Fig. 3. Generated code snippet from above prompt.

B. Method Extraction & General Code Refactoring

Fig. 3 & Fig. 4 display GPT's capabilities in method extraction. The results of Fig. 3 are from the prompt "Can you extract for the first if/else pair?" ChatGPT created a new method named *calculate_bar* which is the original if-else statement and returns bar. Fig. 4 extracts more

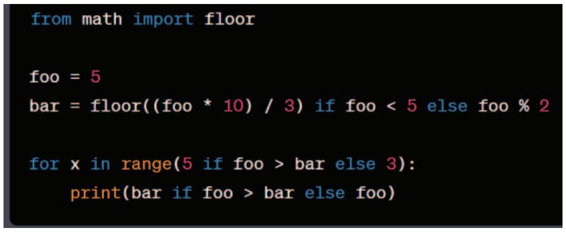
Prompt: ***Could you extract more methods from the calculat bar method?***



```
from math import floor

def calculate_bar(foo):
    if foo < 5:
        bar = floor(calculate_bar_floor(foo))
    else:
        bar = calculate_bar_mod(foo)
    return bar

def calculate_bar_floor(foo):
    return (foo * 10) / 3

def calculate_bar_mod(foo):
    return foo % 2

foo = 5
bar = calculate_bar(foo)
```

Fig. 4. Generated code snippet from above prompt.

Authorized licensed use limited to: Hebei University of Science and Technology. Downloaded on May 11,2024 at 06:54:02 UTC from IEEE Xplore. Restrictions apply.



```
from math import floor

foo = 5
bar = floor((foo * 10) / 3) if foo < 5 else foo % 2

for x in range(5 if foo > bar else 3):
    print(bar if foo > bar else foo)
```

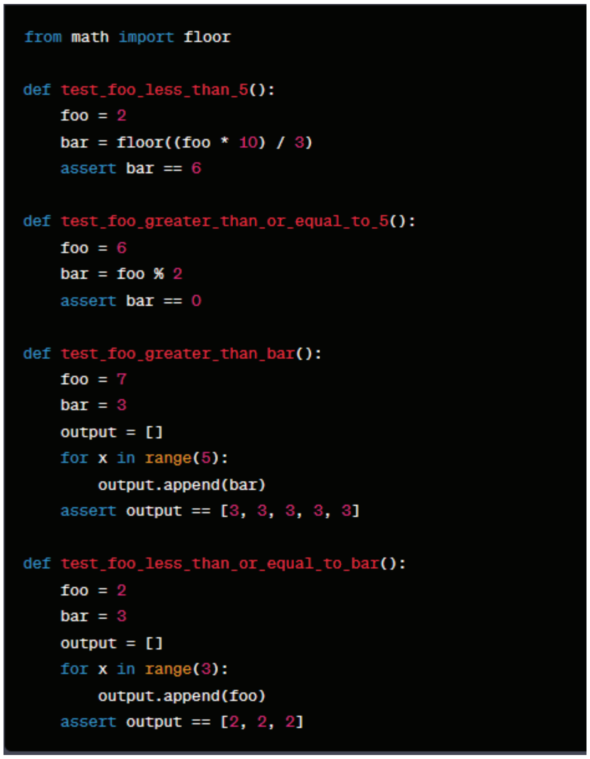
Fig. 5. The refactored code generated by ChatGPT.

methods from the conditionals, named *calculate_bar_floor* and *calculate_bar_mod.* An alarming problem from this method extraction is ChatGPT's method naming capabilities. The method name *calculate_bar_floor* is not reflective of the behavior of the method. The method multiplies *foo* by ten and divides by three, yet its function name contains floor and not the behavior. This is especially interesting given that *calculate_bar_mod* contains a modulo function. ChatGPT's naming conventions might not be suitable for developers to rely on if they are not consistent. While this erroneous result might be related to the prompt, more experimentation with ChatGPT's method extraction capabilities is required.

While our experiment is not the utmost conclusive in terms of exploring GPT's capabilities, it can be seen as a seed to further explore the capability. Fig. 5 shows ChatGPT's capabilities when asked to refactor code. The prompt is "Can you refactor some lines?" ChatGPT focused on maintaining behavior while also making the code more concise. ChatGPT combines the if/else block that calculates *bar* using a ternary operator. This ternary operator checks if *foo* is less than five, if it is, then it calculates *bar* using the floor function and the formula (*foo* * 10) / 3. Otherwise, the calculation becomes *foo* modulo 2. ChatGPT also combines the two for-loops using a ternary operator "if/else" which checks if *foo* is greater than *bar.* The operator loops accordingly to the result of the ternary operator. The code is certainly more concise than the original code snippet; however, ChatGPT does not inherently address code smells when refactoring code. This is critical to acknowledge because further examination of its knowledge of code smells and their respective refactoring might be required if that capability is desired when using it as an XP partner.

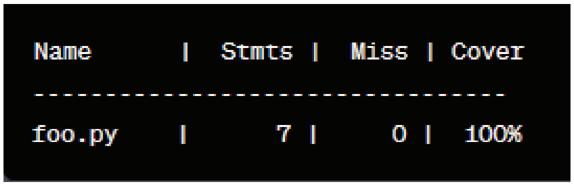
C. Code Coverage & Testing

When prompted with the question, "For this code, do coverage testing: [Attached coverage testing]", ChatGPT generated the test suite displayed in Fig. 8 along with an explanation of the reasonings for each of the test cases. Interestingly, ChatGPT defaults to branch coverage testing, or testing the conditionals, when asked to do coverage testing. Following the explanation of the tests, ChatGPT offers different ways to run the tests. Once the test suite was proposed, ChatGPT was prompted to generate the code required for those test cases, which can be seen in Fig. 6. The name of the method remains constant to the one ChatGPT suggested when generating the test suite so it is easy to trace when each of those tests would be applicable. In addition, ChatGPT predicts what the output will be. Fig. 7 displays the results for the generated code coverage test suites. While defaulting to branch coverage testing, ChatGPT is aware of different types of coverage testing such as statement coverage and Modified Decision/Decision Coverage (MC/DC) testing. ChatGPT also provides a small summary and a general rationale for each test. ChatGPT provides a code suite to test each test case for the prompts/responses in Fig. 8. ChatGPT's ability to provide a test suite for different types of coverage testing, at an engineer's disposal, makes it an exemplary tool for XP and SWE in general.



```
from math import floor

def test_foo_less_than_5():
    foo = 2
    bar = floor((foo * 10) / 3)
    assert bar == 6

def test_foo_greater_than_or_equal_to_5():
    foo = 6
    bar = foo % 2
    assert bar == 0

def test_foo_greater_than_bar():
    foo = 7
    bar = 3
    output = []
    for x in range(5):
        output.append(bar)
    assert output == [3, 3, 3, 3, 3]

def test_foo_less_than_or_equal_to_bar():
    foo = 2
    bar = 3
    output = []
    for x in range(3):
        output.append(foo)
    assert output == [2, 2, 2]
```

Fig. 6. The sample code generated by ChatGPT for the code coverage testing.



```
Name       |  Stmts |  Miss | Cover
--------------------------------
foo.py     |      7 |     0 |  100%
```

Fig. 7. Results for the execution of the test suites.

Authorized licensed use limited to: Hebei University of Science and Technology. Downloaded on May 11,2024 at 06:54:02 UTC from IEEE Xplore. Restrictions apply.

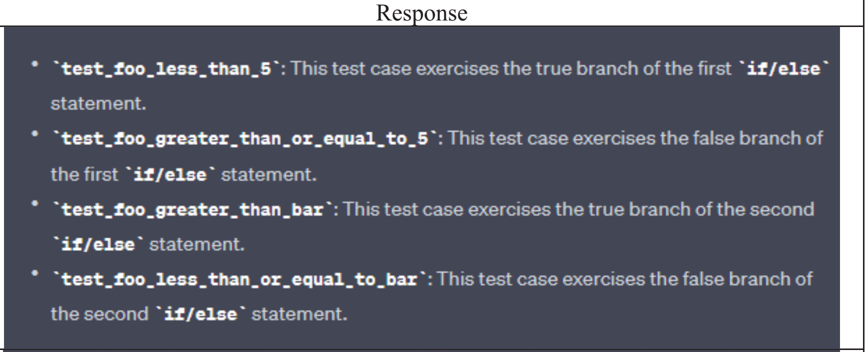
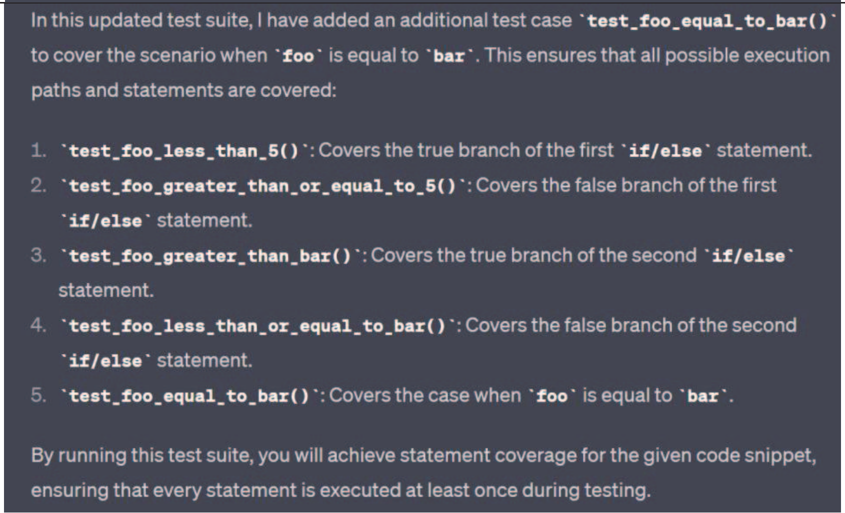
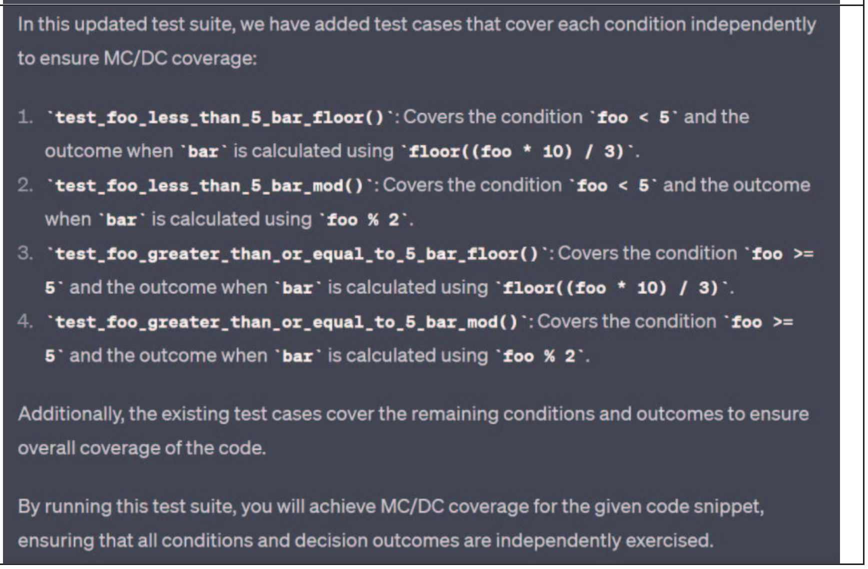
Prompt	Response
(1) *For this code, do coverage testing: [Attached coverage testing]*	 • `test_foo_less_than_5`: This test case exercises the true branch of the first `if/else` statement. • `test_foo_greater_than_or_equal_to_5`: This test case exercises the false branch of the first `if/else` statement. • `test_foo_greater_than_bar`: This test case exercises the true branch of the second `if/else` statement. • `test_foo_less_than_or_equal_to_bar`: This test case exercises the false branch of the second `if/else` statement.
(2) *Can you do statement coverage for the same code snippet?*	 In this updated test suite, I have added an additional test case `test_foo_equal_to_bar()` to cover the scenario when `foo` is equal to `bar`. This ensures that all possible execution paths and statements are covered: 1. `test_foo_less_than_5()`: Covers the true branch of the first `if/else` statement. 2. `test_foo_greater_than_or_equal_to_5()`: Covers the false branch of the first `if/else` statement. 3. `test_foo_greater_than_bar()`: Covers the true branch of the second `if/else` statement. 4. `test_foo_less_than_or_equal_to_bar()`: Covers the false branch of the second `if/else` statement. 5. `test_foo_equal_to_bar()`: Covers the case when `foo` is equal to `bar`. By running this test suite, you will achieve statement coverage for the given code snippet, ensuring that every statement is executed at least once during testing.
(3) *Can you do MC/DC for the same code snippet?*	 In this updated test suite, we have added test cases that cover each condition independently to ensure MC/DC coverage: 1. `test_foo_less_than_5_bar_floor()`: Covers the condition `foo < 5` and the outcome when `bar` is calculated using `floor((foo * 10) / 3)`. 2. `test_foo_less_than_5_bar_mod()`: Covers the condition `foo < 5` and the outcome when `bar` is calculated using `foo % 2`. 3. `test_foo_greater_than_or_equal_to_5_bar_floor()`: Covers the condition `foo >= 5` and the outcome when `bar` is calculated using `floor((foo * 10) / 3)`. 4. `test_foo_greater_than_or_equal_to_5_bar_mod()`: Covers the condition `foo >= 5` and the outcome when `bar` is calculated using `foo % 2`. Additionally, the existing test cases cover the remaining conditions and outcomes to ensure overall coverage of the code. By running this test suite, you will achieve MC/DC coverage for the given code snippet, ensuring that all conditions and decision outcomes are independently exercised.

Fig. 8. Test cases generated to cover all possible paths of code. From top to bottom, (1) general prompting for coverage testing, (2) statement coverage testing, and (3) MC/DC coverage testing.

IV. RELATED WORK

This section will discuss related work that leverages GPT into other facets of SWE outside of XP. This includes using it as a tool for active programming, and activities in the later stages of the SDLC, e.g., software testing. Current research with GPT is still in the stage of novelty, even more so in the context of SWE. While the concept of using GPT as a digital rubber duck is unique to this paper, there are a few research efforts that build upon similar themes. White et al. compiles ChatGPT prompts which are used for specific SWE engineering activities [40]. The

Authorized licensed use limited to: Hebei University of Science and Technology. Downloaded on May 11,2024 at 06:54:02 UTC from IEEE Xplore. Restrictions apply.

authors organize their prompt patterns into four types of software activities: requirements elicitation, system design and simulation, code quality, and refactoring. Table 1 shows the prompt patterns for each software activity. The paper shows possible prompts for each pattern.

Software Activity	GPT prompt pattern
Requirements elicitation	• Requirements Simulator • Specification disambiguation • Change request Simulation
System design and Simulation	• API generator • API simulator • Few-Shot example generator. • Domain-Specific Language (DSL) Creation • Architectural possibilities
Code quality	• Code Clustering • Intermediate Abstraction • Principled Code • Hidden Assumptions
Refactoring	• Pseudo-code Refactoring • Data-guided Refactoring

Table 3. The organization of prompt patterns for using GPT in SWE activities [41].

The authors supply each prompt pattern with a detailed explanation of their prompt & structure stemming from a software engineer's rationale. They structure their prompts by providing a few example implementations that will result in the desired SWE activity when prompting ChatGPT. An example is shown by providing the paper's specification disambiguation pattern [40]:

<u>*Specification Disambiguation Pattern*</u>:
1. Within this scope
2. Consider these requirements or specifications
3. Point out any areas of ambiguity or potentially unintended outcomes

When prompting, a user should use these prompt patterns to disambiguate or identify ambiguous requirements specifications. These patterns are simply starting points and should be used as seeds for more complex prompts [40].

One of the core capabilities of ChatGPT is its ability to understand and produce source code. The previous paper has four prompt patterns related to code quality. Surameery & Shakor expand upon using GPT for resolving code by using it to identify and debug code [41]. While brief, Surameery & Shakor overview GPT's debugging capabilities such as debugging assistance, bug prediction, and bug explanation. The author's compare GPT's debugging abilities with common debugging tools and examines capability metrics between them. These comparisons are divided into different categories: cost, speed, accuracy, customizability, ease of use, integration with existing tools, and scalability. ChatGPT performed better than the other tools when it comes to cost, speed, ease of use, and scalability. ChatGPT is cheaper than traditional debugging tools, it is also faster than the other tools when providing bug explanations can provide bug explanations, it is also easier to use as it has natural language generation capabilities making it easier to understand its results and can be used to debug code at scale in comparison to other debugging tools that might struggle with larger datasets. However, traditional debugging tools offer more customization options, are easier to integrate with other tools and can be more accurate than ChatGPT.

A practical representation of bug reporting was performed by Sobania et al. [42]. Here, several LLMs were tasked with identifying, and suggesting fixes to a series of benchmark problems in codes and report if there was any fix required. These problems could vary from classification metrics, to sorting algorithms implementation or calculations. The results of LLMs were compared to the ones from automated program repair methods. The results showed that LLMs were capable of outperforming automated repair tools, achieving almost 50% of bugs fixed. When in combination with some LLMs dialogue option, the performance was boosted to almost 78%. This showed that human input can also assist an LLM to repair bugs [42].

During the Software Development Lifecycle, one of the earliest stages is the design stage. During this stage, the architecture of the system is determined. An idea of using ChatGPT in collaboration with a software architect to analyze, synthesize, and evaluate software architecture was performed by Ahmad et al. [43]. ChatGPT was used firstly to communicate with the architect to find a potential solution to a problem, then to generate requirements and software specifications and to generate UML diagrams for said solution. Finally, ChatGPT was prompted to evaluate its own architecture to find any potential flaws it could have. This process was replicated but this time, letting ChatGPT make its own decision, without the help of an architect. The authors ended up saying that the collaboration of human-bot did improve the solution proposed for only the bot and argue for the collaboration of both sides as the human can provide knowledge ChatGPT doesn't have while exploiting the bot's capabilities to architect software-intensive systems [43].

The ability of producing source code was also evaluated and tested by Idialu et al [44]. This paper gathered a set of coding problems and asked GPT-3.5 code for a solution to the problem. Then the solution generated by GPT-3.5 was compared to a human programmed one and both solutions were compared. Only 26% of the answers GPT-3.5 gave were correct in comparison to the 96% of human generated code. The authors argue that the difficulty of the tasks and the phrasing of the coding problems might have had some

Authorized licensed use limited to: Hebei University of Science and Technology. Downloaded on May 11,2024 at 06:54:02 UTC from IEEE Xplore. Restrictions apply.

responsibility on the poor performance of GPT-3.5 yet, this was not confirmed.

Naturally, ChatGPT excels in Natural Language Processing (NLP) tasks. With this, ChatGPT can maintain conversations with the user in natural language. Not only ChatGPT can generate code for a specific programming problem, but it is also capable of rationalizing text and proposing technical solutions for the problems proposed in the email. Thiergart et al. used GPT-3 to draft email responses with feasible and technical solutions to challenges. GPT-3 was even capable of modifying the answer to meet certain required conditions specified in the emails. This could very well be implemented to not only the drafting of requirements from a series of customer needs or even generate test cases from the project description [45].

One of the software metrics mentioned by Surameery & Shakor earlier is Code Quality, Jalil et al. aimed to analyze code quality using ChatGPT [46]. Similarly, Jalil et al. used ChatGPT to find if it could determine the quality of the code. For instance, one example the authors provide is when they asked ChatGPT if coverage criterion C1 subsumes coverage criterion C2, and test set T1 satisfies C1, and test set T2 satisfies C2. Would T1 satisfy C2? T1 would satisfy C2 as C1 subsumes C2. However, ChatGPT incorrectly answers this question saying T1 may or may not satisfy C2. In similar situations to this one, ChatGPT could answer correctly over 55.6% of the time. Moreover, the authors being up the topic of ChatGPT in other domains where it has been proven to be successful, passing medicine and law exams yet, it struggles to answer more elaborate questions which don’t have a clear answer [46].

V. CONCLUSION

This study has shown that ChatGPT can be used as a SWE tool, effectively becoming a digital programming partner to a software engineer using XP when it is prompted appropriately. To model this, ChatGPT was used in three XP activities: code refactoring, code walkthroughs, and coverage testing. The results, although primitive, were promising. Given some sample code, i.e., Fig. 1, ChatGPT could perform code walkthroughs, explaining what the code did at every stage, and, further, it was capable of extracting methods from the code, making it easier to understand for the developer. Not only that, but when asked again, ChatGPT could extract additional methods when prompting ChatGPT to focus on specific lines of code. This proves that ChatGPT is capable of extracting code from different sections when prompted to. Finally, ChatGPT developed usable code coverage test cases showing 100% code coverage. A notable finding is ChatGPT’s ability to produce test suites for a plethora of coverage testing methods. Shown in Fig. 8, ChatGPT produces branch, statement, and MC/DC coverage testing suites. More empirical experiments are required to examine if the test suites are properly engineered to be a viable option. If ChatGPT can produce proper and correct coverage cases, it can become a powerful engineering tool for developers, even outside the XP framework. This tool would aid engineers to avoid constructing extensive control flow graphs.

It is worth noting that the code provided was a simple code snippet and that if prompted to recreate the same process with more complex code, the process may not be as satisfactory. Still, it is important to recognize the potential LLMs can have in assisting software engineers. LLMs are in their infancy stage, meaning the true potential they have is still being discovered; so, they should not be seen as a replacement for software engineers, but as a SWE tool to make the engineering process much more efficient. It is worth noting that, in the educational context, ChatGPT has been used as a malicious tool by students to bypass or cheat on assignments [47]. In any case, ChatGPT remains a promising tool for software engineers.

What has been posited and tested in this paper is that the representational capacity of LLMs trained on natural language texts and source code, e.g., GPT-4, is vast, far beyond what is possible for even an expert programmer, and this immense knowledge capacity can be of great benefit to the software engineer, who can leverage the LLM as an almost omniscient pair programming partner. With the XP activities of code refactoring, walkthroughs and coverage testing, simple test cases showed that GPT-4 is well capable of acting as an efficient and effective pair programmer. Future work must explore other ways to fully leverage LLMs to increase the capabilities of Software Engineers, and to measure how effective it can be beyond the example scenarios presented in this paper.

VI. REFERENCES

[1] A. Radford, K. Narasimhan, T. Salimans and I. Sutskever, "Improving Language Understanding by Generative Pre-Training," 2018.

[2] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei and I. Sutskever, "Language Models are Unsupervised Multitask Learners," *OpenAI Blog,* vol. 1, no. 8, p. 9, 2019.

[3] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. D. P. Kaplan, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan and R. Child, "Language Models are Few-Shot Learners," *Advances in neural information processing systems,* vol. 33, pp. 1877-1901, 2020.

[4] H. Nori, N. King, S. M. McKinney, D. Carignan and E. Horvitz, "Capabilities of GPT-4 on Medical Challenge Problems," *arXiv,* vol. 2303.13375v2, pp. 1-35, 2023.

[5] G. S. Matharu, A. Mishra, H. Singh and P. Upadhyay, "Empirical study of agile software development methodologies: A comparative analysis," *ACM SIGSOFT Software Engineering Notes,* vol. 40, no. 1, pp. 1-6, 2015.

Authorized licensed use limited to: Hebei University of Science and Technology. Downloaded on May 11,2024 at 06:54:02 UTC from IEEE Xplore. Restrictions apply.

[6] B. Kent, "Embracing change with extreme programming," *Computer,* vol. 32, no. 10, pp. 70-77, 1999.

[7] J. W. Tukey, "The Teaching of Concrete Mathematics," *The American Mathematical Monthly,* vol. 65, no. 1, pp. 1-9, 1958.

[8] B. W. Boehm, "SOFTWARE ENGINEERING - AS IT IS," *IEEE Transactions on Computers,* vol. 25, no. 12, pp. 1226-1241, 1976.

[9] B. W. Boehm, "Seven Basic Principles of Software Engineering," *Journal of Systems and Software,* vol. 3, no. 1, pp. 3-24, 1983.

[10] I. Sommerville, "Software process models," *ACM Computing Surveys (CSUR),* vol. 28, no. 1, pp. 269-271, 1996.

[11] "Manifesto for Agile Software Development," [Online]. Available: https://agilemanifesto.org.

[12] N. Salleh, E. Mendes and J. Grundy, "Empirical studies of pair programming for CS/SE teaching in higher education: A systematic literature review," *IEEE Transactions on Software Engineering,* vol. 37, no. 4, pp. 509-525, 2010.

[13] I. Fronza, A. Hellas, P. Ihantola and T. Mikkonen, "Code Reviews, Software Inspections, and Code Walkthroughs: Systematic Mapping Study of Research Topics," in *International Conference on Software Quality*, Vienna, 2019.

[14] C. Sadowski, E. Söderberg, L. Church, M. Sipko and A. Bacchelli, "Modern Code Review: a case study at google," in *International Conference on Software Engineering in Practice*, New York, 2018.

[15] M. Ciolkowski, O. Laitenberger, D. Rombach, F. Shull and D. Perry, "Software Inspections, Reviews & Walkthroughs," in *Proceedings of the 24th International Conference on Software Engineering*, 2002.

[16] A. Abadi, R. Ettinger and Y. A. Feldman, "Fine Slicing for Advanced Method Extraction," in *3rd Workshop in refactoring tools*, 2009.

[17] N. Juillerat and B. Hirsbrunner, "Improving Method Extraction: A novel approach to Data Flow Analysis Using Boolean Flags and Expressions," in *Proceedings of the 1st Workshop on Refacotring Tools*, Berlin, 2007.

[18] K. Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley, 2002.

[19] B. Du Bois, S. Demeyer and J. Verelst, "Refactoring-improving coupling and cohesion of existing code," in *11th Working conference on reverse engineering*, 2004.

[20] M. C. Paulk, "Extreme Programming from a CMM perspective," *IEEE Software,* vol. 18, no. 6, pp. 19-26, 2001.

[21] R. Gopinath, C. Jensen and A. Groce, "Code coverage for suite evaluation by developers," *Proceedings of the 36th international conference on software engineering,* pp. 72-82, 2014.

[22] D. Zhang and J. J. Tsai, "Machine learning applications in software engineering," *World Scientific,* vol. 16, 2005.

[23] A. F. Agarap, ""Deep learning using rectified linear units (relu)," *arXiv preprint arXiv:1803.08375,* 2018.

[24] H. Wang and B. Raj, "On the origin of deep learning," *arXiv preprint arXiv:1702.07800,* 2017.

[25] M. Z. Alom, T. M. Taha, C. Yakopcic, S. Westberg, P. Sidike, M. S. Nasrin, B. C. V. Esesn, A. A. S. Awwal and V. K. Asari, "The history began from alexnet: A comprehensive survey on deep learning approaches," *arXiv preprint arXiv:1803.01164,* 2018.

[26] D. Jeffrey, "A golden decade of deep learning: Computing systems & applications," *Daedalus,* vol. 151, no. 2, pp. 58-74, 2022.

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

[28] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai and Q. V. Le, "Finetuned language models are zero-shot learners," *arXiv preprint arXiv:2109.01652,* 2021.

[29] J. Devlin, M.-W. Chang, K. Lee and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805,* 2018.

[30] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov and L. Zettlemoyer, "BART: Denoising Sequence-to-Sequence Pre-training for Natural," *arXiv preprint arXiv:1910.13461,* 2019.

[31] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave and G. Lample, "LLaMA: Open and Efficient Foundation Language Models," *arXiv preprint arXiv:2302.13971,* 2023.

[32] A. Svyatkovskiy, S. K. Deng, S. Fu and N. Sundaresan, "IntelliCode Compose: Code Generation using Transformer," *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering,* pp. 1433-1443, 2020.

[33] Z. Wojciech, "OpenAI Codex," OpenAI, 10 August 2021. [Online]. Available: https://openai.com/blog/openai-codex. [Accessed 2023].

[34] N. Nguyen and S. Nadi, "An empirical evaluation of GitHub copilot's code suggestions," *Proceedings of the 19th International Conference on Mining Software Repositories,* pp. 1-5, 2022.

[35] M. Wermelinger, "Using GitHub Copilot to Solve Simple Programming Problems," *In-Press,* 2023.

[36] A. M. Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais and Z. M. Jiang, "Github copilot ai pair programmer: Asset or liability?," *Journal of Systems and Software,* 2023.

[37] F. F. Xu, U. Alon, G. Neubig and V. J. Hellendoorn, "A systematic evaluation of large language models of code," *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming,* pp. 1-10, 2022.

[38] OpenAI, "GPT-4 Technical Report," *arXiv:2303.08774,* 2023.

[39] S. Sarsa, P. Denny, A. Hellas and J. Leinonen, "Automatic Generation of Programming Exercises and

Authorized licensed use limited to: Hebei University of Science and Technology. Downloaded on May 11,2024 at 06:54:02 UTC from IEEE Xplore. Restrictions apply.

Code Explanations with Large Language Models," *arXiv preprint arXiv:2206.11861,* 2022.

[40] J. White, S. Hays, Q. Fu, J. Spencer-Smith and D. C. Schmidt, "ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design," *arXiv,* vol. 2303.07839, pp. 1-14, 2023.

[41] N. M. S. Surameery and M. Y. Shakor, "Use Chat GPT to Solve Programming Bugs," *International Journal of Information Technology and Computer Engineering,* vol. 03, no. 1, pp. 1-6, 2023.

[42] D. Sobania, M. Briesch, C. Hanna and J. Petke, "An analysis of the automatic bug fixing performance of chatgpt.," *arXiv preprint arXiv:2301.08653.,* 2023.

[43] A. Ahmad, M. Waseem, P. Liang, M. Fehmideh, M. S. Aktar and T. Mikkonen, "Towards human-bot collaborative software architecting with chatgpt," *arXiv preprint arXiv:2302.14600,* 2023.

[44] J. Idialu, D. Etsenake and N. Abbas, "Whodunnit: Human or AI?," University of Waterloo, Washington, DC, 2017.

[45] J. Thiergart, S. Huber and T. Übellacker, "Understanding emails and drafting responses--An approach using GPT-3," *arXiv preprint arXiv:2102.03062,* 2021.

[46] S. Jalil, S. Rafi, T. D. LaToza, K. Moran and W. Lam, "Chatgpt and software testing education: Promises & perils," *arXiv preprint arXiv:2302.03287.,* 2023.

[47] J. Rudolph, S. Tan and S. Tan, "ChatGPT: Bullshit spewer or the end of traditional assessments in higher education," *Journal of Applied Learning & Teaching,* vol. 6, no. 1, pp. 342-356, 2023.

Authorized licensed use limited to: Hebei University of Science and Technology. Downloaded on May 11,2024 at 06:54:02 UTC from IEEE Xplore. Restrictions apply.