LLM-Based Test-Driven Interactive Code Generation: User Study and Empirical Evaluation

Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K. Lahiri

***Abstract*—Large language models (LLMs) have shown great potential in automating significant aspects of coding by producing natural code from informal natural language (NL) intent. However, given NL is informal, it does not lend easily to checking that the generated code correctly satisfies the user intent. In this paper, we propose a novel interactive workflow TiCoder for guided intent clarification (i.e., partial formalization) through tests to support the generation of more accurate code suggestions. Through a mixed methods user study with 15 programmers, we present an empirical evaluation of the effectiveness of the workflow to improve code generation accuracy. We find that participants using the proposed workflow are significantly more likely to correctly evaluate AI generated code, and report significantly less task-induced cognitive load. Furthermore, we test the potential of the workflow at scale with four different state-of-the-art LLMs on two python datasets, using an idealized proxy for a user feedback. We observe an average absolute improvement of 45.97% in the pass@1 code generation accuracy for both datasets and across all LLMs within 5 user interactions, in addition to the automatic generation of accompanying unit tests.**

***Index Terms*—Intent disambiguation, code generation, LLMs, human factors, cognitive load, test generation.**

I. Introduction

Large Language Models (LLMs) have shown tremendous potential in generating natural-looking programs from informal intent expressed in natural language. There has been a surge in research around training LLMs over programming language artifacts in just the last couple of years [1], [2], [3], [4], [5]. Commercial offerings such as GitHub Copilot [6] are widely available, and have been shown to generate a non-trivial fraction of code in real-world scenarios [7].

However, there are several challenges that arise when generating code from natural language specifications [8], [9]. For example, natural language prompts crafted by users may not always fully capture a their intent, as they may contain ambiguous language and lack of nuance. More importantly, it is not possible to automatically evaluate whether code generated from a natural language prompt is correct. Natural language is inherently ambiguous and enforcing the *user intent* through some mechanical process (such as testing, static analysis or formal verification) is not immediately possible.

Consider the following docstring, taken from MBPP [10], a popular Python programming tasks benchmark:

```python
def text_lowercase_underscore(text):
    """Write a function that returns true if the
    input string contains sequences of lowercase
    letters joined with an underscore and false
    otherwise""
```

While the intent may seem obvious at first, it is not immediately clear how to check the correctness of a potential solution. Querying an LLM such as `text-davinci-003` [11] yields several plausibly correct code implementations that pass simple tests such as rejecting the empty string “ ”, or accepting the string `"aa_bb"`. However, it may also produce subtly buggy code solutions that accept strings such as `"aa_bb_cc"`, which is *inconsistent* with the original user intent that expects the string to consist *entirely* of two sequences of lowercase letters joined by an underscore (as defined by the accompanying hidden reference solution and the validation tests from MBPP). In practice, this can often lead to users accepting code with subtle bugs while using LLMs [12], [13]. The apparent ambiguity in this particular docstring, and more importantly the informal nature of natural language, highlights the inability to immediately ascertain the correctness of the code generated by an LLM. Instead, it would be desirable to avoid surfacing such subtly incorrect codes by first clarifying, and partially formalizing, the user intent into a checkable specification.

This issue can be compounded when users are presented with a list of candidate suggestions from LLMs, such as in the Copilot VSCode IDE suggestions pane, which can display up to 10 suggestions. Users often have to linearly scan the list of code suggestions, review them, and reject the incorrect ones until arriving at one that satisfies their intent. In such situations, subtle bugs may be overlooked, with significant downstream impacts. In fact, several recent works exploring developer-AI interaction have highlighted the need for mechanisms to facilitate verification of AI-generated code [9], [14], such as those that allow users to use tests that disambiguate between the different code suggestions [15].

Manuscript received 20 December 2023; revised 30 May 2024; accepted 28 June 2024. Date of publication 22 July 2024; date of current version 19 September 2024. Recommended for acceptance by J. Wang. *(Corresponding author: Sarah Fakhoury.)*

Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri are with the Microsoft Research, Redmond, WA 98052 USA (e-mail: sfakhoury@microsoft.com; saikatc@microsoft.com; shuvendu@microsoft.com).

Aaditya Naik is with the University of Pennsylvania, Philadelphia, PA 19104 USA (e-mail: asnaik@seas.upenn.edu).

Georgios Sakkas is with the University of California, San Diego, CA 92037 USA (e-mail: gsakkas@eng.ucsd.edu).

Digital Object Identifier 10.1109/TSE.2024.3428972

However, prior research has shown that it can be difficult for users to manually provide a sufficient number of test cases to disambiguate suggestions upfront [16].

Inspired by findings around example generation and disambiguation techniques in Programming By Examples (PBE) [17], and recent emerging ability of LLMs to generate tests [18], [19], [20] **in this paper, we propose *leveraging user-feedback* through LLM-generated tests to improve the trust and correctness of LLM-generated code**. Specifically, we propose the workflow of *test-driven interactive code generation* (TiCoder) to (a) clarify (i.e., partially formalize) user intent through generated tests, and (b) generate a *ranked* list of code that is consistent with such tests.

Let us demonstrate a simple instantiation of this framework using the earlier example, where a user prompts an hypothetical LLM to generate code satisfying their natural language intent. Instead of directly displaying a list of plausible code suggestions, our framework TiCoder would query the user with a question:

```
text_lowercase_underscore("aa_bb_cc") == True?
```

Let us assume that the user answers ‘no’, since they expect only two sequences of lowercase letters, joined by one underscore, as mentioned earlier. The workflow would likely query the user again with the following question:

```
text_lowercase_underscore("aa_bb") == True?
```

If the user says ‘yes’, then the system would output the list of approved tests, as well as a set of semantically ranked code suggestions that are consistent with those tests. Once the user chooses a suggestion from such a list, it would generate code along with accompanying tests.

```
def text_lowercase_underscore(text):
    return True if bool(re.search(r'^[a-z]_[a-z]+$
    ', text)) else False
def test_text_lowercase_underscore():
    assert text_lowercase_underscore("aa_bb")==
    True
test_text_lowercase_underscore()
```

In the case of LLM-based code generation, the generated tests not only help make natural language intent more precise and prune incorrect suggestions generated by the LLM, but can also serves as debugging aid for remaining suggestions and regression tests for future code edits [7].

While the proposed framework appears intuitive, it may not scale to more complex code generation tasks. For example, in cases where the user is unable to validate tests, e.g. for tests that require intricate testing frameworks, the a workflow may not be tenable. Furthermore, the utility of the interactive framework is contingent upon (a) the ability of LLMs to generate useful tests, and (b) the cost-benefit trade-off of the overhead of user interaction versus the benefit on pruning and ranking of code suggestions.

To this end, we seek to understand: **How does the proposed workflow impact the performance of developers evaluating AI generated code?** In addition, the proposed framework should scale, augmenting the code generation accuracy of several open and closed-source LLMs. Thus we also seek to answer: **Does proposed workflow augment the accuracy of code generation models?**

To answer these questions, we explore the effectiveness of our proposed framework through a (1) mixed-effects user study and (2) a large scale evaluation of the approach on two Python benchmarks for code generation. **This paper makes the following contributions:**

1) We propose an interactive workflow, TiCoder, for guiding user intent clarification through automatically-generated tests and improving code generation accuracy of LLMs. TiCoder leverages off-the-shelf LLMs for generating code and tests, and provides a mechanism to check AI-generated code through user-approved tests.
2) We evaluate the effectiveness of TiCoder by conducting a mixed-methods user study comparing two different variants of TiCoder for generating and evaluating code suggestions, including a baseline condition representing existing developer-AI interaction workflows. We observe a significant reduction in cognitive effort reported by participants using either variant of TiCoder over existing interaction mechanisms.
3) We further evaluate the performance of the TiCoder workflow at scale by simulating user feedback, using the reference code solution as an idealized proxy. TiCoder is evaluated on on two Python datasets, MBPP and HumanEval, and a mixture of four open and closed sourced LLMs. We demonstrate that TiCoder contributes to improving the code generation accuracy of all LLMs considered. We observe an average absolute improvement of 45.73% in `pass@1` code generation accuracy within 5 user interactions across both benchmarks. In fact, we observe TiCoder can boost smaller model `pass@1` accuracy to levels comparable to much larger models, such as `GPT-4-32k`, within just one user interaction.

II. Related Work

1) Improving Code Generation Accuracy: Techniques for improving code generation accuracy is a rapidly growing field of work. Unlike the work proposed in this paper, these techniques do not consider user feedback, or guide users in clarifying their intent formally; we cover them briefly.

AlphaCode [21] and CodeT [22] both propose techniques to improve code generation accuracy by generating tests using LLMs, and then grouping code suggestions by the set of tests that they satisfy. CodeT [22] refines the approach by scoring tests and code suggestions simultaneously by prioritizing tests that satisfy many code suggestions and prioritizing codes that satisfy many tests. While there are similarities with CodeT in using LLM generated tests to rerank generated code that results in code generation accuracy on benchmarks, TiCoder is complementary as one can apply TiCoder after CodeT. But more importantly, we argue that a user cannot trust the generated code from CodeT any more than using LLM directly. This is because the user is still presented with a set of code suggestions.

In contrast, with TiCoder, we first formalize the user intent through tests allowing the user to constrain the code that the user will need to eventually sample from. TiCoder also allows users to modify test output in one setting, which is not possible in the CodeT approach, where the tests are fixed throughout. As part of future work, we plan to explore if our approach may benefit from code and test ranking algorithms in CodeT.

Similarly, work on program synthesis [23], [24] generates code that satisfies a formal specification either expressed as a logical specification or input-output tests [25]. Our work differs in that we use LLMs to generate code from informal specifications, i.e. natural language intent. However, it would be interesting for future work to leverage user-provided tests to improve the quality of code generation, as explored in recent works [26], [27]. In this work, to evaluate our proposed approach at scale, we simulate user feedback using the code reference implementation as an idealized proxy, similar to prior works in oracle-guided inductive synthesis [15], [28] and interactive program synthesis [29], [30] where an an oracle (reference implementation or users) is queried to identify the output for a given input. However, prior works in this area appeal to an automatic symbolic engine (such as a constraint solver [30] or automata construction [17]) to generate distinguishing example inputs for a pair of programs, which is inconceivable for general purpose imperative programming languages such as Python.

2) Usability of AI Programming Assistants: There exists several prior works exploring the usability of AI programming assistants. In this section, we focus on recent work that identifies challenges related to the expressing of intent and control over the generation suggestions of AI assistants.

Liang et al. [8] identify that *giving up on incorporating generated code*, and *lack of ability to provide feedback*, are the most common usability issues encountered when using completion-based AI programming assistants. This often occurs because the code does not implement the desired functionality, participants do not know why certain code was generated and had trouble controlling the output to be aligned with their desired intent. McNutt et al. [31] enumerate a design space of interactions with code assistants, including how users should be able to disambiguate candidate programs or refine their initial specifications, echoing prior studies have indicated that disambiguation can be valuable in the context of assistants like GitHub Copilot [32] and traditional program synthesis tools [33]. Similarly, Xu et al. [9] explored challenges of IDE-based AI assistants, including how well specified the queries that users formulate are. They find that participants frequently have trouble expressing intent in their natural language queries to the assistant, and issues of under specification often relate to ambiguous instructions, such as omitting variable names.

Mozannar et al. [34] identify 12 core activities associated with using GitHub Copilot and find that programmers often iterate on their prompts until they obtain the suggestion they desire, and spend a significant amount of time verifying code suggestions. In fact, recent work by Bird et al. [14] shows that as result of AI-powered tools, developer roles are shifting so that more time is spent time reviewing code than actually writing code. Several recent works [35], [36], [37] identify clear opportunities for improving the accuracy of LLM code generation techniques. Our work builds upon observations of previous studies, and explores mechanisms to support code evaluation tasks.

III. Research Questions and Paper Organization

We briefly introduce the research questions and discuss paper organization. In the following Section IV-A we introduce our proposed approach: TiCoder. Then we answer two distinct research questions:

RQ1 How does TiCoder impact the performance of python developers evaluating AI generated code, in terms of task correctness, time, and cognitive load? To answer RQ1, we conduct a user study, where participants use AI assistants augmented with the TiCoder workflow. We evaluate the cost benefit tradeoff of the proposed approach on developer effort when evaluating AI generated code.

RQ2 Does the TiCoder workflow improve the accuracy of generated code suggestions? To answer RQ2, we explore the code generation accuracy of LLMs augmented with the TiCoder workflow on two code generation benchmarks in python.

The methodology, evaluation, and results of each research question are organized in the following sections: Sections V and VI describe the methodology and results for RQ1, and Section VII describes the methodology and results for RQ2. We separate methods and results of RQs into distinct sections for clarity. We conclude with a Discussion (Sec. VIII) of the implications of our work to the broader research community, and the Limitations of the presented experiments (Sec. IX).

IV. Proposed Approach: TiCoder

In this section, we outline a proposed workflow for leveraging test generation and user feedback to clarify (i.e., partially formalize) user intent. We refer to this approach as TiCoder (*Test-Driven Interactive Code Generation*), and define two variants of the workflow and surface this interaction to users in the following subsections.

A. High-Level Workflow

Fig. 1 describes the high-level workflow of *Test-Driven Interactive Code Generation* (TiCoder).

1) The user requests the AI programming assistant to generate a function, given optional code context including an existing prefix in a file, a natural language description, and the function header containing method name, parameters and returns.
2) The AI programming assistant internally generates a set of candidate code and test suggestions by prompting an LLM.
3) The set of generated tests are executed for each candidate code suggestion. The set of tests that pass or fail on each code suggestion are stored.
4) Using execution information, the AI programming assistant *ranks* (according to some heuristics) the set of generated tests and then surfaces the top ranked test to the user

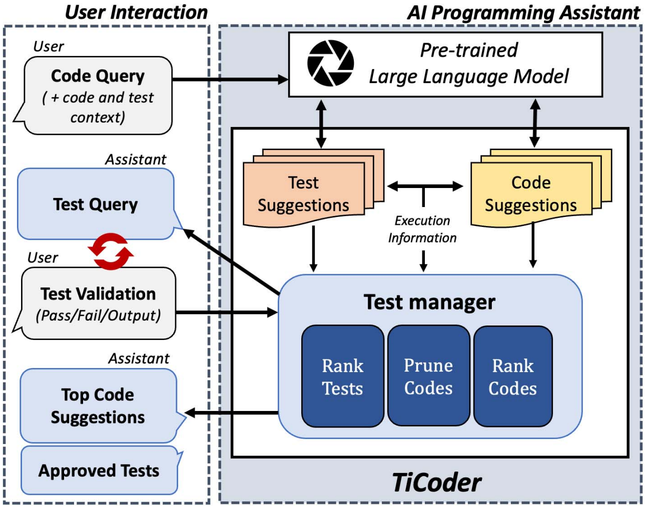


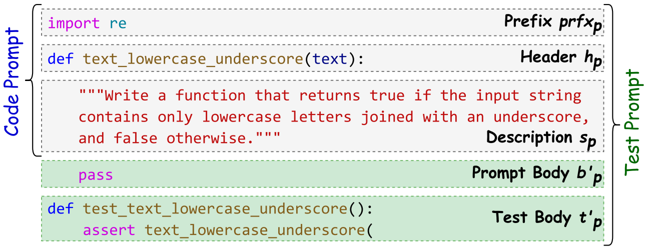
Fig. 1. TiCoder workflow.

as a query; asking the user if a test is consistent with the user's intent.

5) The user responds either PASS, UNDEFINED, or FAIL signifying if the test is respectively: consistent, precondition-violating[1], or inconsistent with the user intent. Optionally, in the case of FAIL, the user can provide the correct test output OUTPUT.
6) The AI programming assistant leverages the user response to prune, and rank the set of code and test suggestions.
7) Interaction steps 4-6 can be repeated for multiple iterations, until a predefined termination criteria (e.g., fixed number of steps, absence of tests) has been satisfied.
8) Once the interaction terminates, the AI programming assistant outputs (a) a set of tests that the user has approved or specified, and (b) a ranked list of code suggestions that are consistent with the user responses.

We define two variants of the workflow: TiCoder-PassFail and TiCoder-Output. The first scenario represents the case where the user provides only a Boolean PASS,FAIL response. The second scenario, TiCoder-Output, extends the first scenario and represents the case where the user provides the expected output OUTPUT in the case of a FAIL test.

We present both the scenarios as they enjoy complementary benefits. The TiCoder-PassFail scenario is more lightweight, in terms of user feedback, as well as, generalizes well for richer tests beyond input-output examples. For example, tests for stateful APIs comprises of a test-prefix as input and the output oracle consists of a non-trivial predicate (e.g., checking functional correctness of a stack object using the predicate `s.pop() == a` on a stack object `s` and element `a`) [19]. On the other hand, TiCoder-Output puts less burden on an LLM to create the correct output for a given test input; relying instead on the user. However, it may require the user to specify a possibly non-trivial test oracle when used beyond input-output examples.



```
import re
def text_lowercase_underscore(text):
    """Write a function that returns true if the input string
    contains only lowercase letters joined with an underscore,
    and false otherwise."""
    pass
def test_text_lowercase_underscore():
    assert text_lowercase_underscore(
```

Fig. 2. Example format, as well as *code* and *test prompts* for the running example.

B. TiCoder Implementation

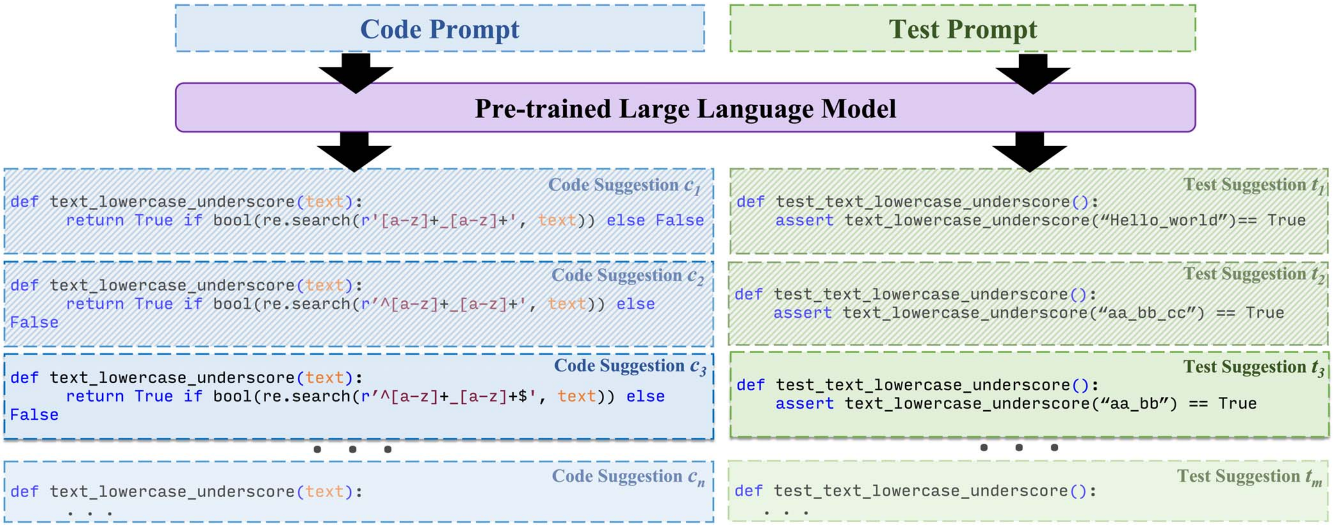
In this section, we discuss one possible implementation of the TiCoder workflow. Specifically, we outline the approach to generating code and test suggestions, ranking candidate tests to surface to the user, pruning and ranking code suggestions by user response. To simplify the presentation, we restrict ourselves to the case of single function synthesis, where the user input consists of a natural language comment s_p, the function header h_p, as well as any optional prefix $prfx_p$ needed to generate the body of the function p. Fig. 2 shows an example for our running example. In addition, we also assume the presence of a set of *hidden* tests T_p (input-output pairs for simplicity) to evaluate the correctness of the generated code, as well as a *hidden* reference (oracle) implementation of p, namely b_p. Our workflow does not have access to either T_p or b_p.

1) Generating Code and Tests: We outline one possible choice for implementing the prompt generation for generating code and test suggestions for an example.

Fig. 2 presents a possible *code prompt* (in the gray boxes) that can be used to query an LLM to produce a set of *code suggestions* for our running example. Querying a LLM with the code generation prompt will result in a set of *code suggestions* similar to ones shown in Fig. 3. Code suggestion c_3 is a valid solution to the problem, while c_1 is an incorrect code suggestion (since it allows the first substring to start with an uppercase letter) and c_2 is also incorrect (since it allows more than one sequence of lowercase letters joined with an underscore). Similarly, the green boxes in Fig. 2 shows one possible *test prompt* that augments the *code prompt* with the statement `pass` as the method body (corresponding to a placeholder implementation in Python) along with the assertion to be completed within a test function. We use the generated test suggestions (Fig. 3) to present the user with a set of tests. Some of these are *consistent* with the user intent (t_3); while others are *inconsistent* with the user intent (t_1 and t_2).

2) Ranking Test Suggestions: After obtaining the set of tests produced by an LLM, the user is presented with a sequence of tests. The user response to these proposed tests in both TiCoder scenarios (TiCoder-PassFail, TiCoder-Output) are used to prune and rank code suggestions. To minimize the number of user interactions, it is desirable to prioritize tests that would result in the most number of incorrect code suggestions being pruned away [15], [29]. To achieve this, the set of tests are executed against the set of possible code suggestions generated by the LLM.

[1] A test violates a precondition if the function is undefined on the test input. For example, the test `assert SquareRoot(-4) == -2` undefined on negative numbers.



```
def text_lowercase_underscore(text):
     return True if bool(re.search(r'[a-z]+_[a-z]+', text)) else False
```

```
def text_lowercase_underscore(text):
     return True if bool(re.search(r'^[a-z]+_[a-z]+', text)) else
False
```

```
def text_lowercase_underscore(text):
     return True if bool(re.search(r'^[a-z]+_[a-z]+$', text)) else
False
```

```
def text_lowercase_underscore(text):
     . . .
```

```
def test_text_lowercase_underscore():
    assert text_lowercase_underscore("Hello_world")== True
```

```
def test_text_lowercase_underscore():
    assert text_lowercase_underscore("aa_bb_cc") == True
```

```
def test_text_lowercase_underscore():
    assert text_lowercase_underscore("aa_bb") == True
```

```
def test_text_lowercase_underscore():
  . . .
```

Fig. 3. *Code* and *test suggestions* for the running example in Fig. 2 generated from a LLM. Code suggestion c_3 and test suggestion t_3 are both *correct*, while code suggestions c_1, c_2 and test suggestions t_1, t_2 are *incorrect* (appear shaded), *i.e.* they don't satisfy the *problem prompts* in Fig. 2.

Then, using this execution information, we adopt a `discriminative` test ranking policy that prioritizes tests that can discriminate best among the set of code suggestions generated by the LLM. If a test t can discriminate between code suggestions well (i.e., splits the set of code suggestions into roughly equal halves), then it would prune away a substantial fraction of the code suggestions irrespective of the user response (either PASS or FAIL).

More precisely, let U be the set of test suggestions and G be the set of code suggestions that have not been pruned away after $k \geq 0$ user interactions. For each test $t \in U$, we split the set of code suggestions G into the sets G_t^+ and G_t^- of code suggestions that pass and fail the assertion in t, respectively. Note that we ignore codes that results in a crash on a test t instead of failing with an assertion failure. We treat these as precondition violation. We then prioritize tests where the ratio of the sizes of these two set is closest to 1. In other words, we rank the tests in decreasing order using the following scoring metric s_{discr}:

$$s_{discr}(t) = \left\{ \begin{array}{ll} 0 & \text{if } \mathtt{max}(|G_t^+|, |G_t^-|) \text{ is } 0 \\ \mathtt{min}(|G_t^+|, |G_t^-|)/ & \\ \quad \mathtt{max}(|G_t^+|, |G_t^-|) & \text{otherwise} \end{array} \right\}$$

Note that the test ranking strategy is uniform for both the scenarios, although the test output will be possibly mutated by the user response in TICODER-OUTPUT.

Consider the example in Fig. 3. Consider the two tests t_1 and t_2: Two code suggestions $\{c_2, c_3\}$ `FAIL` on test suggestion t_1 while one suggestion $\{c_1\}$ `PASS`, making $s_{discr}(t_1) = \mathtt{min}(1,2)/\mathtt{max}(1,2) = 1/2$. Similarly, two code suggestions $\{c_1, c_2\}$ `PASS` on test suggestion t_2 while one suggestion $\{c_3\}$ `FAIL` and $s_{discr}(t_2) = 1/2$. All code suggestions in this example `PASS` on test t_3 making $s_{discr}(t_3) = 0$.

3) Pruning and Ranking Code Suggestions: TICODER returns a ranked list of code suggestions, whose behavior is consistent with all the user responses, and prunes the other code suggestions generated by the LLM, whose execution behavior on tests is contradictory to user expectation. Let us first consider the case of code pruning. Let $t \doteq (i, o)$ be a test in the form of an input-output example presented to the user. If the user responds PASS, then we prune any code $c \in G$ for which executing $c(i) \neq o$. Similarly, if the user responds FAIL, then we prune any code $c \in G$ for which executing $c(i) = o$. In addition, for TICODER-OUTPUT if the user provides the desired output o' for the input i, then we can further prune any code suggestion c for which $c(i) \neq o'$. Note that we cannot soundly prune any code if the user responds with UNDEFINED.

Finally, we define a simple code ranking strategy that uses the tests in U to determine a ranking on code suggestions in G as follows: Each generated code $c \in G$ is executed with each test $t \in U$ and gets assigned as a score *the number of passing tests* d_c. The codes are then ranked based on the decreasing order of d_c.

Following from the example in the previous section, represented in Fig. 3, code suggestion c_1 passes on all tests $\{t_1, t_2, t_3\}$, code suggestion c_2 passes on $\{t_2, t_3\}$ and code suggestion c_3 passes on $\{t_3\}$. Our ranking would therefore rank c_1 highest initially in the absence of any feedback from the user.

V. RQ1: USER STUDY METHODOLOGY

We aim to understand how the TICODER workflow may support software developers as they use AI-programming assistants to generate and evaluate code suggestions. We are seeking to answer the following research question:

RQ1 How does TiCoder impact the performance of python developers evaluating AI generated code, in terms of task correctness, time, and cognitive load?

To answer our research question we conduct a controlled study with 15 participants consisting of 3 coding evaluation tasks. To complete each task, participants are asked to interact

TABLE I
*WEEKLY DENOTES A FEW TIMES A WEEK, *MONTHLY DENOTES A FEW TIMES A MONTH

ID	Python Experience	Python Frequency	AI Programming Assistant Use	Occupation
Pilot	>5 years	Daily	Daily	Industry
Pilot	>5 years	Monthly	Monthly	Industry
Pilot	>5 years	Daily	Daily	Industry
P1	>5 years	Monthly	Daily	Industry
P2	>5 years	Weekly	Monthly	Industry
P3	>5 years	Rarely or never	Rarely or never	Industry
P4	>5 years	Daily	Daily	Industry
P5	3−5 years	Weekly	Weekly	Academia
P6	>5 years	Weekly	Weekly	Academia
P7	>5 years	Weekly	Monthly	Industry
P8	>5 years	Monthly	Rarely or never	Academia
P9	>5 years	Daily	Monthly	Academia
P10	1−2 years	Weekly	Daily	Academia
P11	>5 years	Daily	Daily	Academia
P12	3−5 years	Weekly	Rarely or never	Academia
P13	>5 years	Weekly	Rarely or never	Academia
P14	3−5 years	Rarely or never	Rarely or never	Academia
P15	>5 years	Daily	Daily	Industry

with one of the following AI assistants: Assistant 1 with no user intent refinement, Assistant 2 representing TICODER-PASSFAIL workflow, or Assistant 3 representing TICODER-OUTPUT workflow. Participants use each assistant to generate and evaluate a set of code suggestions.

We recruit participants using a mix of distribution lists and personal contacts. 3 of 18 participants were used as part of the pilot study to inform our design, and the remaining 15 are used in the final experiment. Table I contains participant demographic information. 8 participants are either professional software engineers or researchers at Microsoft, and the remaining 10 participants are PhD students from academia. The study was IRB approved with voluntary participation and paid $15. All interviews were conducted over a video-conferencing platform and lasted approximately 45-minutes.

Participants were asked to complete each code evaluation tasks with one of the three different AI code generation assistants. Each task had a time limit of 15 minutes. We use a within subject design, such that *each participant uses all three assistants*, i.e. a different assistant for each task. Each AI assistant represents one treatment under study, which we describe in the next subsection.

A. Treatments

The experiment includes one control condition and two distinct treatment conditions, implemented as different AI programming assistants. Each assistant differs in it's interaction mechanism with the developer and dictates the method in which to surface the final set of code suggestions shown to each user. To ensure that the same set of codes is shown to all participants across treatments, we pre-select the prompt used to generate code suggestions. Second, to ensure we measure the impact of our dependent variables on only the process of evaluating AI generated code, we also restrict the ability to edit the AI generated code suggestions. The interaction framework of each Assistant is described below:

1) Control Condition: AI Programming Assistant 1: Assistant 1 represents the control condition for the experiment. Given the pre-selected prompt, Assistant 1 generates 5 code suggestions for the user, surfaced in a random order. Participants using Assistant 1 always see 5 unique code suggestions. We make this decision to reflect the current user experience scenario of several real-world AI code generation tools, such as GitHub Copilot's completion panel. For example, the GitHub Copilot completion panel in VSCode shows the user up to 10 possible code suggestions at a time. Our decision is also informed by research pointing to the benefit of surfacing multiple code suggestions [8], [31], [32], [33]. We limit the maximum number of codes to 5 so as to allow the participant to complete each task within 15 minutes.

2) Treatment condition: AI Programming Assistant 2: Assistant 2 represents the TICODER-PASSFAIL (Sec. IV-A) scenario, where a user provides instructions in the form of a prompt, and then the Assistant generates test cases that the user must validate. The user validates each test by indicating if the test should pass or fail. Assistant 2 then uses the tests to prune any of the 5 code suggestions that differ in behaviour validated by the user. For example, if the user decides that the test should pass, only codes that pass the test are retained. These retained code suggestions are shown to the user, in random order.

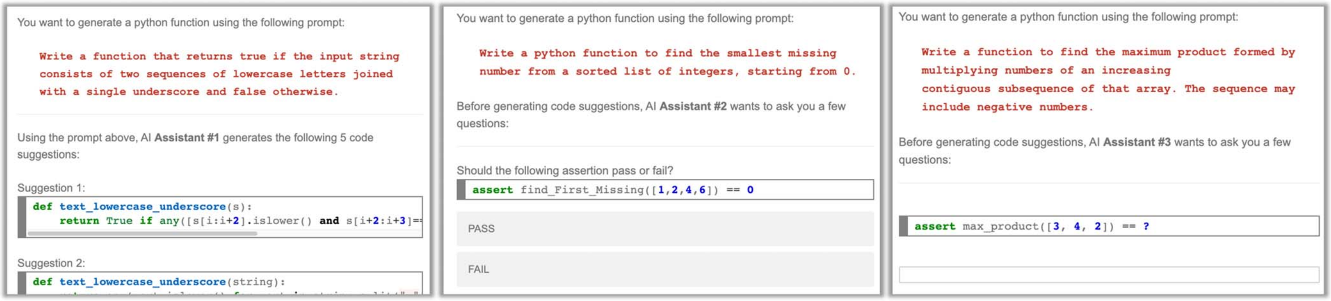
3) Treatment Condition: AI Programming Assistant 3: Assistant 3 represents the TICODER-OUTPUT scenario (Sec. IV-A). Instead of indicating whether a test should pass/fail (i.e. Assistant 2 interaction mechanism), users must provide the expected output of the test. Assistant 3 then uses the tests completed by the user to prune any of the 5 code suggestions that do not generate an output consistent with what the participant defined.

Both Assistants 2 and 3 use the tests to prune away generated codes that do not match the behaviour specified by the participant. We restrict the number of pruned codes in each task so that a participant using Assistant 2 or 3 will always see between 3-4 code suggestions if they correctly evaluate the tests shown to them. We make this decision to reflect the potential real-world scenario where TICODER is able to prune away at least 1 of the candidate code suggestions generated by an AI Assistant. However, a participant may see less than 3 code suggestions if they specify a contradictory or incorrect program behaviour through their answers to the tests.

Participants interact with the assistants in an online survey platform, but they are able to copy and paste the generated code and tests into an IDE of their choice during the task. All code is formatted so as to not introduce external factors into the participants' time. The interactive nature of the AI Assistants in encoded into the survey logic, to mimic real-world execution and pruning of code suggestions based on a user's answers. Participants can maintain their view of the tests they validated in the survey throughout the task.

TABLE II
TASKS INCLUDED IN THE USER STUDY, DERIVED FROM THE MBPP DATASET. A1 REPRESENTS THE CONTROL TREATMENT, A2 REPRESENTS THE TICODER-PASSFAIL TREATMENT, AND A3 REPRESENTS THE TICODER-OUTPUT TREATMENT

Task	Task Name	Description	Treatments
T1	LOWERUNDERSCORE	Write a function that returns true if the input string consists of two sequences of lowercase letters joined with a single underscore and false otherwise.	A1, A2, A3
T2	FIRSTMISSING	Write a function that finds the smallest missing number from a sorted list of integers, starting from 0.	A1, A2, A3
T3	MAXPRODUCT	Write a function to find the maximum product formed by multiplying numbers of an increasing contiguous subsequence of that array. The sequence may include negative numbers.	A1, A2, A3



You want to generate a python function using the following prompt:

```
Write a function that returns true if the input string
consists of two sequences of lowercase letters joined
with a single underscore and false otherwise.
```

Using the prompt above, AI **Assistant #1** generates the following 5 code suggestions:

Suggestion 1:

```
def text_lowercase_underscore(s):
    return True if any([s[i:i+2].islower() and s[i+2:i+3]==
```

Suggestion 2:

```
def text_lowercase_underscore(string):
```

You want to generate a python function using the following prompt:

```
Write a python function to find the smallest missing
number from a sorted list of integers, starting from 0.
```

Before generating code suggestions, AI **Assistant #2** wants to ask you a few questions:

Should the following assertion pass or fail?

```
assert find_First_Missing([1,2,4,6]) == 0
```

PASS

FAIL

You want to generate a python function using the following prompt:

```
Write a function to find the maximum product formed by
multiplying numbers of an increasing
contiguous subsequence of that array. The sequence may
include negative numbers.
```

Before generating code suggestions, AI **Assistant #3** wants to ask you a few questions:

```
assert max_product([3, 4, 2]) == ?
```

Fig. 4. From left to right: examples of different interaction sequences invoked by assistant 1 on task T1 (directly display all code suggestions), assistant 2 on T2 (validate the test output on a given input), and assistant 3 on T3 (specify the output for a given input).

B. Task Design

We selected coding tasks that would satisfy the following criteria, for each of the three tasks: (1) evaluating 5 AI-generated code suggestions could be completed in fifteen minutes, (2) there are syntactically valid but semantically incorrect code completions given by the LLM (`GPT-3.5`) with a diversity of error types across tasks (3) they varied in problem domain and complexity, and (4) the LLM could generate reasonable tests that capture the diverse error types.

1) Identifying Task Candidates: We select task candidates from the MBPP dataset [38], a popular code generation benchmark, consisting of short Python functions designed to be solved by entry-level programmers. MBPP provides a natural language instruction, a set of tests, and a ground truth code implementation for each problem. We cluster functions from MBPP based on problem domain, complexity as measured by cyclomatic complexity and size of the function in terms of lines of code. From each cluster we identified a set of candidate functions for which we generated a code and test completions for using a LLM. We finally selected 3 problems for the code completion tasks that best satisfied the selection criteria. These problems represent three distinct styles: MAXPRODUCT is an algorithmic task involving dynamic programming, LOWERUNDERSCORE involves using regex for string manipulation, and FIRSTMISSING involves a recursive binary search. The tasks are detailed in Table II.

2) Generating Code and Test Suggestions: To generate the code and test candidates, we give the natural language instructions from the MBPP dataset as a prompt to the OpenAI `GPT-3.5-turbo` chat completion endpoint with the default API parameters (temperature = 1.0). We then sample a set of 5 incorrect codes using the tests from the MBPP dataset, to identify buggy programs. We also run the set of generated tests against the set of codes to make sure at least 1 and at most 2 code suggestions are caught by the test, to restrict the number of codes that would be pruned away. Rather than manually inject bugs into the ground truth program, we choose to sample the set of buggy codes from the LLM to reflect the nature of bugs users may encounter in AI generated code.

The final set of tests and codes are fixed per task, regardless of the treatment used. For each task there are 5 suggestions: 4 buggy codes and 1 code that is extracted as the ground truth from the MBPP dataset. If the ground truth program extracted from MBPP does not handle certain pre-condition violations, we augment the code to match the task intent. For Assistants 2 and 3, we show exactly 2 of the AI generated tests for each task. The final set of codes are either directly shown to the user by Assistant 1, or first pruned based on the user's evaluation of the tests for Assistants 2 and 3.

C. Study Protocol

At the start of the study, participants are given general instructions around how to interact with each AI assistant, the differences between them, and how to validate generated tests. Participants are also given time to set up their Python interpreter or environment before the start of the study. The survey interface used to interact with the AI assistants is shown in Fig. 4. Participants are able to view the coding task description, and depending on the treatment received, they can answer the AI Assistant's question, around the validation of a test case, directly in the survey. Once code suggestions are surfaced by the Assistant, participants are allowed to copy the code and run it for debugging, along with the set of provided tests, depending on the treatment. For each task, participants were asked to identify if the AI Assistant had returned a correct code suggestion, and if yes, which one.

We employ a Latin Square Design to systematically vary the pairing of tasks and AI assistants. Each participant completes three tasks (T1, T2, and T3), each with a different AI assistant (Assistant 1, Assistant 2, Assistant 3). The order in which the participants use the AI assistants for each task is randomized to account for learning effects. This design ensures that each AI assistant is used an equal number of times across all tasks and positions, thus balancing potential order effects and providing a robust comparison of the AI assistants

For each task, participants are encouraged to ask any questions around the task instructions. Our aim is to approximate the scenario where the user clearly understands what they want the AI Assistant to generate, such that they would query the AI Assistant with the same or similar prompt originally used to generate code and test suggestions. Although the real world usage of the workflow would differ, as developers often edit their prompts, we choose to fix the prompt to control for the generated code and tests across participants. Furthermore, we are not interested in the task of code generation, rather *code validation*, i.e. not if the user can edit the prompt to get different suggestions, but rather how the TiCoder workflow can help refine user provided natural language specification through tests, and how it may impact a developer's ability to validate code, and locate a correct suggestion out of a set of generated codes. While TiCoder may help reduce the number of times a user must edit their original prompt, we save this exploration for future work.

D. Measured Variables

From the study recordings and user-submitted survey data, we collect a set of metrics on each task completed by the participants:

Time. We measured time taken to complete each task from the recordings of each participant interview. Time for each task includes time taken to evaluate any tests. We measure time on task to determine if the TiCoder workflow adds significant time overhead due to validation of tests, as compared to the control condition.

Correctness. The correctness is dependant on 1) the correct evaluation of the generated tests, relative to the oracle code implementation and 2) the correct selection of the code suggestion whose behaviour reflects the intent of the prompt. Each task has at most 1 correct answer: either one of the generated code suggestions is correct or none of suggestions are correct.

Cognitive Load. The TiCoder workflow aims to improve code generation accuracy and reduce the number of candidate code snippets that a user needs to review, ultimately reducing the cognitive effort of code evaluation. However, TiCoder may also add additional cognitive effort, stemming from the effort required to validate tests. To test the impact of the workflow on cognitive effort, after each task we measure participants' self-reported cognitive load via their responses to five NASA TLX questions [39], a standardized approach to measuring self-reported cognitive load used widely across disciplines [40]. We measure the following metrics using the standard 20 point scale: mental demand, effort, perceived success, pace, and stress.

TABLE III
Mixed-Effects Model Analysis Results for Control (Assistant 1) and Treatment (Assistant 2, 3) Conditions. (* Denotes a Significant Observation. – Indicates No Significance.)

Metric	**Assistant 1 (Mean)**	**Assistant 2 (Mean)**	**Assistant 3 (Mean)**	**Pairwise Significance**
Correctness* (0,1)	0.40	0.84	0.64	$a1 < a2 (p = 0.001)$
Time (seconds)	327.7	284.15	253.88	-
Cognitive Load* (0-100)	45.46	28.00	29.52	$a1 > a2 (p = 0.007)$ $a1 > a3 (p = 0.012)$
Mental Demand* (0-20)	12.5	7.50	7.6	$a1 > a2 (p = 0.001)$ $a1 > a3 (p = 0.004)$
Stress* (0-20)	8.26	3.84	6.35	$a1 > a2 (p = 0.02)$
Pace* (0-20)	8.13	5.38	4.70	$a1 > a3 (p = 0.04)$
Confidence (0-20)	13.5	15.92	15.88	-
Effort* (0-20)	11.00	7.15	6.7	$a1 > a2 (p = 0.02)$ $a1 > a3 (p = 0.014)$

E. Evaluation of Measured Variables

For each measured variable, time, correctness, and dimensions of cognitive load, we run a mixed-effects regression model. We use with either linear or logistic models depending on the data type. We use the treatment condition as the fix-effects independent variable, and participant ID and coding task as the random-effects variables.

We conduct an omnibus test using ANOVA to calculate the p-value of the treatment condition (the assistant used) against the measured metrics. To correct for multiple comparisons and conduct False Discovery Rate (FDR) correction [41] for significant pairs of conditions. We only report Omnibus p-values for pairs of conditions for which the results are statistically significant, and the direction of significance. We chose mixed-effects models to account for individual variability (participants) and hierarchical data structures (task treatment pairs), using ANOVA for omnibus to test the significance of fixed effects (the assistant used), and conducted FDR to provide a comprehensive assessment of treatment effects while controlling for Type I errors.

VI. RQ1: User Study Results

Our key quantitative results are summarized in Table III. The last column of Table III provides Omnibus p-values for pairs of conditions for which the results are statistically significant.

A. Impact on Task Correctness

Using the mixed-effects regression model with the correctness of the task (coded as 0 or 1), as the dependent variable: The mean correctness was 0.40 for participants using Assistant 1, 0.84 Assistant 2, and 0.64 Assistant 3. Although the mean is higher in Assistant 2 and 3, the effect is significant for

Assistant 2 only with (p = 0.001). Looking at the set of mistakes made by participants, we notice several interesting observations. In general, participants using Assistant 1 are less likely to identify the correct code suggestion from the set of 5 suggestions.

For example, for Task 1, 3/4 participants that failed to identify the correct suggestion were using Assistant 1. Looking at their responses, all 3 participants identified different suggestions as correct. One participant, **P7** chose to not execute any of the code suggestions, while the other two participants **P3, P5** did write tests to evaluate the code suggestions, they were not able to find a test to characterize the bug. Similarly, for task 3, 2/4 participants that failed to identify the correct suggestion were using Assistant 1, and chose different candidate suggestions. Interestingly, both participants also only tested a subset of the codes, based on an initial guess of the correct suggestion.

Looking at the differences between Assistant 2 and 3, we notice that mistakes stem from both incorrect evaluations of the surfaced tests and incorrect evaluation of the code suggestions. For example, in Task 2 FIRSTMISSING, the first test case surfaced to participants by Assistant 2 is shown shown in Fig. 2b: `assert find_First_Missing([1,2,4,6]) ==0`. All participants shown this test correctly answered that this test should pass. However, when the output `== 0` is obfuscated on the same test by Assistant 3, 50% of the participants indicated that the test should evaluate to 0, and the other 50% indicated that it should (incorrectly) evaluate to 3. It is interesting to note that given a correct test by Assistant 2, participants are able to correctly evaluate it, however, if Assistant 2 had generated an incorrect test output (== 3) it may not always be the case that participants are able to catch this bug.

However, not all tests surfaced by Assistant 2 are correct. In Task 1, both tests surfaced by Assistant 2 had incorrect test outputs; testing edge case scenarios that should fail. For example `text_lowercase_underscore("Hello_world") == True`. For Assistant 2, all participants were able to correctly identify that the test should fail. For participants using Assistant 2, 4/5 indicated that it should evaluate to 'False' while one participant indicated that it should (incorrectly) evaluate to 'True'.

In Task 3, all participants using Assistant 2 and Assistant 3 were able to correctly evaluate the tests surfaced by the Assistants. However, 2 of the participants using Assistant 3 were not able to identify the correct code suggestion, whereas all participants using Assistant 2 were successful.

By construction, upon the erroneous evaluation of a test case by a user, the TiCoder workflow will prune all valid programs that pass on the test. Therefore, participants that incorrectly evaluate a test case will no longer see any valid AI-generated programs and cannot correctly complete the task, unless they specify that none of the code suggestions are correct. In the TiCoder workflow, noisy user response guarantees that generated code does not match the 'ground truth' user intended specifications. Therefore, in practice, the option to skip a test evaluation is imperative to the usability of the workflow, and to reduce uncertainty as the source of noisy input by the user. Though TiCoder may significantly support users in correctly evaluating code suggestions, the potential for noisy feedback is a critical risk to consider.

Key Findings Participants using TiCoder Assistant 2 are significantly more likely to correctly evaluate AI generated code. Participants using Assistant 3 were, on average, more likely to correctly evaluate code suggestions compared to participants that were not using TiCoder. However, participants were also more prone to making mistakes while providing test outputs that dealt with edge cases.

B. Impact on Task Time

To test the effect of each Assistant on time, we used a mixed-effects regression model, with time as the dependent variable. The mean time taken by participants using Assistant 1 is 327.7 seconds, 284.15 for Assistant 2, and 253.88 for Assistant 3. Although the means differ slightly across Assistants, on average participants using TiCoder take less time to complete the code evaluation tasks. However, this effect is not significant.

This indicates that the additional overhead of requesting participants to verify or provide the output for a test case does not add significantly to the time taken to complete the task. The time taken to evaluate code suggestions may be tempered by the number of code suggestions pruned, and the fact that Assistants 2 and 3 provide test cases to support the code evaluation process. One indicator of how long a participant takes to complete a task may be tied to their code evaluation strategy. We notice that, regardless of the treatment, participants that choose to execute and test every single code suggestion, take much longer than participants that scan the code suggestions and selectively execute and test candidate suggestions that 'look' correct to them.

For example, (**P2**) had relatively longer task times when using all 3 assistants, and chose to mentally execute every suggestion, identify the bug in each suggestion, and then proceeded to programmatically execute and test their hypothesis. Due to their thorough evaluation strategy, **P2** was correct on all tasks. However, we do not observe a correlation between time on task and correctness, both within, and across tasks (Pearson's Correlation Coefficient $r = 0.016, p = 0.911$).

Key Findings The time taken to validate test cases, introduced by TiCoder, does not introduce significant overhead to total task time. Participants using TiCoder take, on average, less time to complete the code evaluation tasks, however, this effect is not significant.

C. Impact on Task Induced Cognitive Load

We analyze the self-reported cognitive load of participants across 5 dimensions, outlined by the NASA TLX: mental demand, effort, pace, stress, and confidence of the task correctness. Cognitive load is reported as the cumulative sum across all 5 dimensions. Using a mixed-effects regression model with the cognitive load as the dependent variable, we observe that participants using Assistants 2 and 3 report significantly less cognitive load. Looking more closely at the different dimensions, for Assistant 2 participants report significantly less mental demand, stress, and effort required to complete the task. For Assistant 3 participants report significantly less mental demand, effort, and better pace.

Overall, we posit that this effect might be observed due to the the reduced number of code suggestions that the user must evaluate, and that tests serve as concrete mechanisms for which to reason about the code; as well as provide a starting point for more extensive testing of the candidate functions, making it easier to get the task started. For example, when asking clarifying questions about the prompt used in a task, participants using Assistant 1 struggle to articulate their question before coming up with an illustrative test case. For Task 3 MAXPRODUCT, participants using Assistant 1 had difficulty conceptualizing 'increasing contiguous subsequence'. The interviewer made sure to answer any questions the participant had, but took care to not give concrete examples to not bias the participant. For example, **P19** first asked "*so you're multiplying just two numbers, but it has to be next to each other?*". When the interviewer clarified that it could be more than two numbers, given that the sequence is increasing, the participant articulated their question with an example "*...so if I have, 1 2 4 1, it would be 1 by 2 by 4?*". In contrast, participants that had similar questions, but were using Assistants 2 or 3, were able to more easily articulate their questions using test cases generated by the AI Assistant.

Key Findings Participants using TiCoder, in both Assistant 2 and 3 settings, report significantly less task-induced cognitive load while evaluating AI generated code. This effect may be explained by the code pruning and test clarification mechanisms offered by TiCoder.

VII. RQ2: BENCHMARK EVALUATION

Results from our user study, Section 5, indicate that TiCoder can significantly improve correctness of participants evaluating AI generated code, and that the workflow helps to reduce task-induced cognitive load. However, it is unclear if the proposed workflow is able to effectively generate tests that, once validated, can prune and rank a set of code suggestions with higher accuracy, on a large set of problems. To evaluate the potential utility of the TiCoder workflow at scale, we implement TICODER-PASSFAIL and TICODER-OUTPUT, and conduct an empirical evaluation on two state-of-the-art benchmarks for code generation in python. We aim to answer the following research question:

RQ2 Does the TiCoder workflow improve the accuracy of generated code suggestions?

A. Datasets

We use two Python programming datasets for our evaluation, including the *sanitized* version of the *MBPP dataset* [38], dataset from Google, and the *HumanEval dataset*, introduced in the Codex paper [1], to answer the research questions. *MBPP* consists of 427 and *HumanEval* of 164 examples in the format described in Sec IV-B, along with the hidden tests and reference implementations. We modify the original *HumanEval* dataset to remove any (non-hidden) input-output examples that are included in the docstring to avoid making the test generation task trivial.

B. Evaluation Metric

For evaluating the *correctness of the generated code suggestions*, we use the popular metric `pass@k` for evaluating the accuracy of code-generation by LLMs with respect to the hidden tests provided by each dataset [1]. A code suggestion is correct if it passes all the hidden tests, and `pass@k` determines the mean *expected* value of an arbitrary sample of size k to contain at least one correct solution. To evaluate TiCoder, we define the metric `pass@k@m` to denote the *ranked*`pass@k` for the code suggestions after $m \geq 1$ user queries. Recall that TiCoder outputs a ranked list of code suggestions, so `pass@k@m` measures if any of the top k code suggestions is correct. Given that the list of code suggestions from TiCoder are ordered, our metric `pass@k@m` is not a statistical measure (unlike `pass@k` which measures the statistical odds of any sample of size k containing a correct code solution), but deterministically check if any one of the top k ranked code suggestions is correct.

C. Models and Baselines

TICODER augments AI assistant workflows to improve the code generation accuracy of the underlying LLM. To assess TICODER's benefits across various LLMs, we've chosen four state-of-the-art completion models, which include a mix of closed-source and open-source models. We provide a brief description of each model next.

- ***OpenAI `code-davinci-002`, `text-davinci-003`*** [1] is a closed source LLM specifically designed for code completion tasks. It is based on the GPT-3 architecture containing 175B parameters.[2] `text-davinci-003` is also a closed source model of size 175B parameters, however, it is based on the GPT-3.5 architecture and InstructGPT [11] and can be used for a variety of natural language tasks. Compared to other non-chat based completion models, `text-davinci-003` demonstrates highly competitive performance on a number of tasks.
- ***OpenAI `GPT-3.5-turbo`, `GPT-4-turbo`, `GPT-4-32k`*** The OpenAI chat models are based on the pretrained GPT-3 model, which is fine-tuned using Reinforcement Learning with Human Feedback (RLHF) [11]. While these models are not explicitly fine-tuned for code generation, they have demonstrated strong capabilities on several related tasks [42], [43]. We use OpenAI APIs for the `gpt-3.5-turbo`, `gpt-4-32k`, and `gpt-4-turbo` endpoints.
- ***Salesforce `CodeGen-6B`, `CodeGen2.5-7B`*** [3] is an open source LLM, with 6B parameters, trained specifically to translate natural language instructions to

[2] Access to this model was removed to the public by OpenAI in March 2023, but continues to be made free and available to researchers upon request.

TABLE IV
MODEL BASELINE AND TICODER RESULTS FOR TWO PYTHON DATASETS: MBPP AND HUMANEVAL. USER INTERACTION RESULTS ARE SIMULATED FOR UP TO M=5 TEST EVALUATION INTERACTIONS. WE HIGHLIGHT, IN BLUE, THE HIGHEST ACCURACY IN EACH COLUMN

Dataset	Model	Baseline **pass@k**		TICODER-PASSFAIL **pass@1@m**					TICODER-OUTPUT **pass@1@m**				
		1	100	1	2	3	4	5	1	2	3	4	5
MBPP	`text-davinci-003`	49.16	86.88	68.04	75.26	77.33	77.88	78.08	77.00	82.20	83.38	83.59	83.75
	`code-davinci-002`	48.25	89.75	68.42	76.21	79.37	81.18	81.97	76.97	85.51	87.10	87.56	87.71
	`CodeGen-6B`	14.85	69.55	28.62	37.91	45.18	49.97	53.67	39.64	54.01	62.34	65.30	66.56
	`CodeGen2.5-7B`	28.32	84.74	50.27	59.70	65.02	67.84	69.58	62.78	73.84	78.10	79.52	80.42
	`GPT-3.5-turbo`	61.91	84.77	75.12	77.31	78.21	79.04	79.03	78.24	80.86	81.06	81.76	81.78
	`GPT-4-turbo`	69.80	86.88	80.71	81.90	82.38	83.20	83.12	83.91	84.63	84.97	85.65	85.65
	`GPT-4-32k`	67.13	87.35	81.56	82.62	82.97	83.11	83.70	84.78	85.28	85.31	85.40	85.79
HumanEval	`text-davinci-003`	44.13	87.80	60.70	67.54	71.41	72.18	72.81	65.02	75.48	78.04	79.34	80.18
	`code-davinci-002`	30.49	91.49	51.66	62.65	70.30	73.11	74.37	57.25	74.5	81.99	83.70	84.50
	`CodeGen-6B`	11.41	43.55	15.32	19.29	24.64	28.11	29.56	26.34	32.81	39.08	44.23	48.12
	`CodeGen2.5-7B`	21.39	76.21	32.82	41.03	46.51	49.47	52.33	35.86	51.60	61.54	65.02	68.25
	`GPT-3.5-turbo`	59.45	89.02	73.16	76.48	77.01	77.75	79.22	73.44	76.60	78.45	78.45	79.51
	`GPT-4-turbo`	62.62	89.63	78.36	80.42	80.92	80.84	81.34	82.22	83.17	83.42	83.62	83.84
	`GPT-4-32k`	60.72	89.02	76.10	78.43	79.07	79.48	79.49	81.37	82.46	82.49	82.49	82.54

code. `CodeGen2.5-7B` [44] is an improvement on `CodeGen-6B` and is slightly larger, with 7B parameters. Currently, this model is the state-of-the-art for code generation compared to other models of similar parameter size.

Our aim is to understand how TiCoder can help improve code generation accuracy across different LLMs, and not to identify the best performing model. Therefore, we use default configurations for each model, and only alter temperature. We experimented with different temperature configurations to optimize performance and diversity of generated code and test suggestions. Intuitively, a temperature closer to 1 allows LLMs to provide a more diverse set of solutions, whereas a temperature closer to 0 forces LLMs to only generate fewer solutions with the highest confidence. Following [1], [45], [46] for all models we settle on a temperature of 0.8, as it maximizes the number of examples for which at least one correct code is produced within 100 suggestions for $k > 1$ in $pass@k$. To account for the non-determinism of the LLM generations, for each dataset, we only query each model once to generate an initial set of 100 code and 50 test suggestions into a *cache* of responses. We use the same cache across all experiments that involve the specific LLM.

D. Simulating User Response

Our proposed workflow requires real-time user response to determine if a generated test is consistent with the user's intent. Therefore, in order to evaluate TICODER *offline* with large-scale benchmark datasets, we define a proxy to *simulate* real-time user response.

Similar to oracle-guided inductive synthesis [15], [28], [29], we use the reference implementation b_p as an *oracle* to answer if a test (i, o) is consistent with the user intent, and provide the expected output $b_p(i)$ when the test output o does not match the user intent (for TICODER-OUTPUT). In other words, we assume that the intent of the user is precisely captured by the semantics of the (hidden) reference implementation. Further, if a test input crashes the reference code, we treat the user response as UNDEFINED to model a precondition violation. However, this models an *idealized* user interaction because, in practice, users may sometimes be unable to answer a test query in a reasonable amount of time (say, when asked about the value of the 100th prime number). As observed in the results of the user study, unlike an idealized user, real participants may sometimes provide noisy input. For example, we observe that participants are more prone to making mistakes while providing test outputs that dealt with edge cases. Therefore, using the oracle as a proxy indicates that our empirical evaluation represents an *upper bound* on the improvement that TiCoder can have on the benchmarks with real users.

E. Results

To answer RQ2, we evaluate the performance of four different models, with and without TiCoder in the TICODER-PASSFAIL and TICODER-OUTPUT settings on MBPP and HumanEval datasets. Table IV contains all results for each model.

The first column contains the baseline `pass@1` and `pass@100` for each model on MBPP and HumanEval datasets. Note that `pass@100` denotes the fraction of examples for which an LLM generates at least one correct code suggestion within 100 suggestions. The second and third columns contain the results for each model, augmented with TiCoder in TICODER-PASSFAIL and TICODER-OUTPUT settings respectively. We report the `pass@1@m` metric, with m, the number of test-validation user interactions, ranging from 1 to 5. We report `pass@k@m` only for the case of $k = 1$ as it is the strictest setting for assessing the impact of TiCoder. TiCoder improves the accuracy of `pass@k@m` for higher values of k as well, but we do not present them in the interest of space.

All three chat models, `GPT-3.5-turbo`, `GPT-4-turbo` and `GPT-4-32k` demonstrate the highest accuracy for `pass@1` with comparable performance across datasets. As expected, `text-davinci-003` and `code-davinci-002`, the two largest completion models, achieve the fourth and fifth baseline performance on both datasets. However, `code-davinci-002` achieves the highest `pass@100` across all models and both datasets.

Overall, we observe that both TiCoder in the TiCoder-PassFail and TiCoder-Output settings significantly improve `pass@1` performance, across all models. As the number of test validation queries increase from $m = 1$ to $m = 5$ we also observe consistent improvement in `pass@1` performance. Although the improvement is most pronounced at $m = 1$, compared to baseline.

For example, on MBPP, TiCoder-PassFail improves `pass@1` baseline performance of `text-davinci-003` from 49.16% to 68.04%, an absolute improvement of 18.88% within one user query. TiCoder-Output improves performance to 77.00%, which is an absolute `pass@1` improvement of 27.84% within one user query. This increases to 38.55% with 5 queries. While smaller models achieve lower pass@1 and pass@100 accuracy, TiCoder still provides modest boosts in accuracy. For the worst performing model, `CodeGen-6B` on HumanEval, TiCoder-PassFail provides an absolute `pass@1` improvement of 3.91% within one interaction, and TiCoder-Output provides an absolute improvement of 14.93%.

In fact, we observe that TiCoder can boost code generation accuracy of smaller models to comparable performance of much larger SOTA LLMs. For example, after just one user interaction `code-davinci-002` on MBPP achieves 68.42% accuracy in the TiCoder-PassFail setting, which is in fact *higher* than the `pass@1` accuracy of all three SOTA chat models: `GPT-4-32k`, `GPT-4-turbo`, and `GPT-3.5-turbo`.

Finally, as expected TiCoder-Output consistently provides higher accuracy compared to TiCoder-PassFail, since the former allows users to fix the incorrect test output. TiCoder-Output achieves an average absolute improvement of 45.73% in the code generation accuracy for both datasets and across all LLMs within 5 user interactions. However, it is worth noting that TiCoder-PassFail, even with its lightweight feedback (that generalizes to richer tests or specifications), always stays within 9% of the benefits of TiCoder-Output. This demonstrates the power of LLMs to generate test cases that satisfy user intent.

Key Findings TiCoder significantly improves pass@1 performance for all studied LLMs on both benchmarks, with performance improvements increasing with every test validation interaction. TiCoder can boost small model performance within one user interaction, outperforming pass@1 accuracy of larger models like GPT-4-32k. Additionally, the lightweight TiCoder-PassFail scenario always stays within 9% of the performance of TiCoder-Output even in this idealized simulated user setting.

VIII. Discussion

A. Tests as a Developer-AI Disambiguation Mechanism

Results of our user study indicate that using tests as an interactive mechanism to formalize user intent, and then prune and rank AI generated code suggestions, can meaningfully improve programmer performance. Results show that both Assistant 2 (TiCoder-PassFail) and Assistant 3 (TiCoder-Output) are statistically distinguishable from Assistant 1 (the control condition without TiCoder), where no interactive test case verification or code pruning is used. Participants using Assistant 2 are significantly more likely to correctly evaluate AI generated code suggestions, and report reduced task induced cognitive load, without negative impact on time to complete each task. However, compared to Assistant 2, participants made more mistakes when validating tests with Assistant 3.

Further research to explore a trade-off between the approaches, in practice, should be considered. However, we find that surfacing test cases in both forms might serve as a helpful mechanism for which to reason about generated code. While it is true that correctness likely depends on the evaluation strategies used by each participant, participants that were shown test cases by the AI Assistants performed significantly better.

Recent work has shown that current developer-AI interaction workflows, as simulated by Assistant 1 in our study, have raised new issues in the way that developers write code. Results suggest a need for interaction mechanisms that can support disambiguation; a critical feature of the usability of AI-Assistants [31], [32]. We observed how TiCoder can surface a ranked list of tests that delineate the space of possible code suggestions, providing a concrete mechanism for identifying potential ambiguities in natural language used to prompt LLMs. Furthermore, recent work has shown that developers spend significant amount of time verifying code suggestions [14], [34]. While we do not observe a statistically significant impact on the time taken to evaluate AI-generated code suggestions when using TiCoder, we do observe a significant reduction in the amount of cognitive effort required. We hypothesize that tests, which provide tangible artifacts for developers to reason about the code, can serve as a facilitating mechanism for constructing mental models of code functionality. Ultimately, this supports developers in the task of code evaluation. Future work should explore this in more detail.

B. Improving LLM Code Generation Capabilities With Verified Test Cases

Results of our benchmark evaluation indicate that, across all models studied, both implementations of TiCoder, TiCoder-PassFail and TiCoder-Output, can be used to augment the accuracy of an LLM through improved ranking and pruning. Specifically, we observe that using tests to better constrain the space of possible code suggestions can improve pass@k accuracy. This highlights the fact that current LLM capabilities may not be fully realized in practice: when prompted for multiple code suggestions, LLMs often are capable of generating a correct answer, but mechanisms to better rank the set of suggestions is needed. However, the performance boost provided by TiCoder is contingent on a set of high quality tests used by `discriminative` test ranking policy (Sec. IV).

If the underlying LLM is unable to generate high quality tests, the ranking and pruning mechanism may not be as helpful. Future work should explore more sophisticated mechanisms for

TABLE V
Results of Prompting GPT-4 With Validation Tests in the Prompt

		TiCoder-PassFail	Tests in Prompt			
Dataset	Model		Single (pass@k)		All (pass@k)	
		pass@1@1	k= 1	k = 5	k = 1	k = 5
MBPP	`GPT-4-32k`	81.56	78.26	87.81	80.88	91.58
HumanEval	`GPT-4-32k`	76.10	65.25	81.37	70.39	86.99

generating high quality tests that capture important specifications about the code. In this work, we explore the user study scenario where TiCoder prunes away some, but not all of the AI generated code suggestions.

It is worth noting that if the code suggestions generated by the underlying LLM predominantly exhibit consistent behavior, TiCoder can still be valuable to a user by providing meaningful tests alongside the code suggestions. For example, in a scenario where all of the suggestions are correct with respect to the user's intent, TiCoder may not prune away any of the code suggestions, but provides some guarantees about program behaviour to the user.

C. The Value of Execution Based Pruning

The TiCoder workflow helps users by automatically generating and ranking interesting test cases, instead of requiring the user to manually write them. However, users may have a set of test cases already in mind when querying a LLM. We explore how such a scenario compares to the TiCoder workflow by adding the set of validation tests, i.e. the ground truth test set provided in each dataset, to the code generation prompt. We instruct `GPT-4-32k` to generate a code suggestion that passes on the set of validation tests provided in the prompt. We then evaluate the $pass@k$ accuracy at $k = 1$ and $k = 5$ and compare to the TiCoder$pass@1@m$ accuracy from Table IV. Table V contains the result of this experiment, showing both experiments where a single test is added in the prompt, as well as an experiment where all available validation tests are added in the prompt, the exact number of tests varies across both datasets.

On both MBPP and HumanEval we observe a boost in $pass@k$ accuracy over the baseline prompt used Table IV. For example, on MBPP `GPT-4-32k` achieves a $pass@1$ of 67.13% with the baseline prompt (no tests) and 78.26% when including a single test in the prompt. This increases to a $pass@1$ of 80.88% when all tests are added in the prompt. As expected, this demonstrates that adding tests in the prompt does help improve model performance. However, recalling the $pass@1@1$ accuracy of the TiCoder-PassFail, within one user feedback loop, accuracy reaches 81.56%, out performing the $pass@1$ accuracy where a single test is included in the prompt by 3.31%. TiCoder even performs slightly better (0.68%) compared to the case when all tests are in the prompt.

This indicates that even if the user supplies all test, there is no guarantee that the underlying LLM will fulfill the user intent and generate a code suggestion that passes on the tests. Conversely, with only one user interaction on a highly distinguishing test, obtained by ranking LLM generated tests, code generation accuracy greatly improves, *matching the scenario where several tests are provided to the LLM without the added burden on the user to manually come up with test cases.*

D. Considerations for AI-Generated Tests: Precondition Violations

Tests generated by LLMs may contain pre-condition violating inputs that would cause the reference implementation to crash or fail, and would result in a Undefined response from the user, resulting in no pruning of code.

As an illustrative case, consider from MBPP, the example of a reference implementation with (implicit) precondition that the argument `nums` array is non-empty, and throws a `divisionByZero` at the return statement for an empty `nums` array.

```python
from array import array
def zero_count(nums):
    """Write a function to find the ratio of zeroes
     to non-zeroes in an array of integers."""
    n = len(nums)
    n1 = 0
    for x in nums:
        if x == 0:
            n1 += 1
        else:
            None
    return n1/(n-n1)
```

The following test is ranked highest by the discriminative ranking strategy:

```python
assert zero_count([]) == 0, "Empty List"
```

For this test, out of the (deduplicated) 80 code suggestions from `code-davinci-002`, 8 suggestions pass the test, 11 suggestions fail the test and 61 suggestions crash on this test. The score for this test is 8/11 = 0.73, which is the highest among all tests. However, this test does not lead to any code pruning as the user responds Undefined in both the TiCoder-PassFail and TiCoder-Output scenarios since the empty array causes the reference implementation above to fail. Thus, our results in RQ2 account for tests with Undefined responses, reflecting the possible real world impact of pre-condition violating tests on the accuracy boost provided by TiCoder.

IX. Limitations and Threats

Generalizability of user study results. We evaluate TiCoder under highly controlled experimental conditions, and the ability of developers to validate tests in more complex code generation scenarios may not scale. Our study explores two distinct test validation mechanisms surfaced in the TiCoder workflow: TiCoder-PassFail and TiCoder-Output. On the selected tasks, we observe that, in general, participants were able to successfully evaluate tests in both interaction scenarios. However, in practice specifying the output of generated tests may not always be a straightforward or simple task. In addition, we restrict participants abilities to edit the code prompt and code suggestions, to control for variables across participants. However, this is not a true reflection of real-world interaction behaviours. Future work should explore the impact of TICODER on developer productivity in real-world code settings, with broader audiences. In addition, we only explored how TICODER impacts the correctness of code evaluation, i.e. how well users can disambiguate code suggestions. For example, a future experiment might examine how TICODER impacts online metrics such as code acceptance rates, or the total proportion of code contributed by the AI system, accommodating for solutions that provide partial correctness.

Generalization of benchmark evaluation results. We also empirically evaluate TICODER using two popular and state-of-the-art research Python benchmarks for code generation tasks: MBPP and HumanEval. While both benchmarks exercise common programming patterns, they may not be representative of real-world software development. Our findings may not generalize to a different set of programs across different languages and problem domains.

Test execution overhead. The proposed TICODER workflow incurs the cost of additional LLM inference, to generate candidate tests, as well as resource costs related to executing tests for generated code suggestions. Cost of execution may be non-trivial, and might not scale in scenarios where users wish to use an AI assistant to generate complex code. Nevertheless, the potential reliability guarantees and reduced effort for code verification represent a valuable trade-off when weighed against the costs of inference and execution in real-world scenarios. Future work should examine practical use of TICODER at scale.

X. CONCLUSION

In this work, we propose the workflow of test-driven interactive code generation using LLMs, and study it's effectiveness through a user study and empirical evaluation on code generation benchmarks. Our findings provide encouraging results around guiding user intent clarification for generating more correct programs.

In future work, we plan to extend and evaluate our implementation reflecting real-world scenarios including: more complex programs, an in-situ user study for various software development tasks, and an empirical evaluation on realistic benchmarks such as CoderEval [47] and NL2Fix [43]. Finally, we plan to explore if TICODER can be extended to richer forms of formal specifications beyond tests, such as property based tests or pre- and post-conditions generated from user-defined prompts [48].

REFERENCES

[1] M. Chen et al., "Evaluating large language models trained on code," 2021, *arXiv:*2107.03374.

[2] A. Chowdhery et al., "PaLM: Scaling language modeling with pathways," 2022, *arXiv:2204.02311*.

[3] E. Nijkamp et al., "A conversational paradigm for program synthesis," 2022, *arXiv:2203.13474*.

[4] D. Fried et al., "InCoder: A generative model for code infilling and synthesis," 2022, *arXiv:2204.05999*.

[5] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proc. 6th ACM SIGPLAN Int. Symp. Mach. Program. (MAPS)*, New York, NY, USA: ACM, 2022, pp. 1–10, doi: 10.1145/3520312.3534862.

[6] "Github copilot," GitHub. Accessed Aug. 5, 2022. [Online]. Available: https://github.com/features/copilot/

[7] A. Ziegler et al., "Productivity assessment of neural code completion," in *Proc. 6th ACM SIGPLAN Int. Symp. Mach. Program. (MAPS@PLDI)*, San Diego, CA, USA, S. Chaudhuri and C. Sutton, Eds., New York, NY, USA: ACM, 2022, pp. 21–29, doi: 10.1145/3520312.3534864.

[8] J. T. Liang, C. Yang, and B. A. Myers, "Understanding the usability of AI programming assistants," 2023, *arXiv:2303.17125*.

[9] F. F. Xu, B. Vasilescu, and G. Neubig, "In-IDE code generation from natural language: Promise and challenges," *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, vol. 31, no. 2, pp. 1–47, 2022.

[10] J. Austin et al., "Program synthesis with large language models," 2021, *arXiv:2108.07732*.

[11] L. Ouyang et al., "Training language models to follow instructions with human feedback," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 35, 2022, pp. 27730–27744.

[12] O. Asare, M. Nagappan, and N. Asokan, "Is GitHub's copilot as bad as humans at introducing vulnerabilities in code?" 2022, *arXiv:2204.04741*.

[13] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with AI assistants?" 2022, *arXiv:2211.03622*.

[14] C. Bird et al., "Taking flight with copilot: Early insights and opportunities of ai-powered pair-programming tools," *Queue*, vol. 20, no. 6, pp. 35–57, 2022.

[15] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng. (ICSE)*, Cape Town, South Africa, J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, Eds., New York, NY, USA: ACM, 2010, pp. 215–224, doi: 10.1145/1806799.1806833.

[16] T. Lau, "Why programming-by-demonstration systems fail: Lessons learned for usable AI," *AI Mag.*, vol. 30, no. 4, pp. 65–65, 2009.

[17] T. Zhang, L. Lowmanstone, X. Wang, and E. L. Glassman, "Interactive program synthesis by augmented examples," in *Proc. 33rd Annu. ACM Symp. User Interface Softw. Technol.*, 2020, pp. 627–648.

[18] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *Proc. 45th Int. Conf. Softw. Eng. (ICSE)*, 2023, pp. 919–931.

[19] E. Dinella, G. Ryan, T. Mytkowicz, and S. Lahiri, "TOGA: A neural method for test oracle generation," in *Proc. Int. Conf. Soft. Eng. (ICSE)*, New York, NY, USA: ACM, May 2022, pp. 2130–2141. [Online]. Available: https://www.microsoft.com/en-us/research/publication/toga-a-neural-method-for-test-oracle-generation/

[20] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "Adaptive test generation using a large language model," 2023, *arXiv:2302.06527*.

[21] Y. Li et al., "Competition-level code generation with alphacode," 2022, *arXiv:2203.07814*.

[22] B. Chen et al., "Codet: Code generation with generated tests," 2022, *arXiv:2207.10397*.

[23] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," *Found. Trends Program. Lang.*, vol. 4, nos. 1–2, pp. 1–119, 2017, doi: 10.1561/2500000010.

[24] A. Solar-Lezama, "The sketching approach to program synthesis," in *Programming* Languages and Systems, Z. Hu, Ed., Berlin, Germany: Springer-Verlag, 2009, pp. 4–13.

[25] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *Proc. ACM Sigplan Notices (PoPL'11)*, Austin, Texas, USA, 2011, pp. 317–330. [Online]. Available: https://www.microsoft.com/en-us/research/publication/automating-string-processing-spreadsheets-using-input-output-examples/

[26] N. Jain et al., "Jigsaw: Large language models meet program synthesis," in *Proc. Int. Conf. Softw. Eng. (ICSE)*, 2022, pp. 1219–1231. [Online]. Available: https://www.microsoft.com/en-us/research/publication/jigsaw-large-language-models-meet-program-synthesis/

[27] K. Rahmani et al., "Multi-modal program inference: A marriage of pre-trained language models and component-based synthesis," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–29, 2021, doi: 10.1145/3485535.

[28] S. Jha and S. A. Seshia, "A theory of formal synthesis via inductive learning," *Acta Informatica*, vol. 54, no. 7, pp. 693–726, 2017, doi: 10.1007/s00236-017-0294-5.

[29] V. Le, D. Perelman, O. Polozov, M. Raza, A. Udupa, and S. Gulwani, "Interactive program synthesis," 2017, *arXiv:1703.03539*.

[30] R. Ji, J. Liang, Y. Xiong, L. Zhang, and Z. Hu, "Question selection for interactive program synthesis," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI)*, New York, NY, USA: ACM, 2020, pp. 1143–1158, doi: 10.1145/3385412.3386025.

[31] A. M. McNutt, C. Wang, R. A. Deline, and S. M. Drucker, "On the design of ai-powered code assistants for notebooks," in *Proc. CHI Conf. Human Factors Comput. Syst.*, 2023, pp. 1–16.

[32] S. Barke, M. B. James, and N. Polikarpova, "Grounded copilot: How programmers interact with code-generating models," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, pp. 85–111, 2023.

[33] M. Mayer et al., "User interaction models for disambiguation in programming by example," in *Proc. 28th Annu. ACM Symp. User Interface Softw. Technol.*, 2015, pp. 291–301.

[34] H. Mozannar, G. Bansal, A. Fourney, and E. Horvitz, "Reading between the lines: Modeling user behavior and costs in AI-assisted programming," 2022, *arXiv:2210.14306*.

[35] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Proc. Extended Abstr. CHI Conf. Human Factors Comput. Syst. (CHI EA '22)*, New York, NY, USA: ACM, 2022, doi: 10.1145/3491101.3519665.

[36] N. Nguyen and S. Nadi, "An empirical evaluation of GitHub copilot's code suggestions," in *Proc. 19th Int. Conf. Mining Softw. Repositories*, 2022, pp. 1–5.

[37] S. Imai, "Is GitHub copilot a substitute for human pair-programming? An empirical study," in *Proc. ACM/IEEE 44th Int. Conf. Softw. Eng.: Companion Proc.*, 2022, pp. 319–321.

[38] J. Austin et al., "Program synthesis with large language models," 2021, *arXiv:2108.07732*.

[39] S. G. Hart and L. E. Staveland, "Development of NASA-TLX (task load index): Results of empirical and theoretical research," in *Advances in Psychol.*, vol. 52. Amsterdam, The Netherlands: Elsevier, 1988, pp. 139–183.

[40] S. G. Hart, "Nasa-task load index (NASA-TLX); 20 years later," in *Proc. Human Factors Ergonomics Society Annual Meeting*, vol. 50, no. 9. Los Angeles, CA, USA: Sage, 2006, pp. 904–908.

[41] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: A practical and powerful approach to multiple testing," *J. Roy. Statist. Soc.: Ser. B (Methodol.)*, vol. 57, no. 1, pp. 289–300, 1995.

[42] T. X. Olausson, J. P. Inala, C. Wang, J. Gao, and A. Solar-Lezama, "Demystifying GPT self-repair for code generation," 2023, *arXiv:2306.09896*.

[43] S. Fakhoury, S. Chakraborty, M. Musuvathi, and S. K. Lahiri, "Towards generating functionally correct code edits from natural language issue descriptions," 2023, *arXiv:2304.03816*.

[44] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, "CodeGen2: Lessons for training llms on programming and natural languages," 2023, *arXiv:2305.02309*.

[45] R. Li et al., "StarCoder: May the source be with you!" 2023, *arXiv:2305.06161*.

[46] B. Roziere et al., "Code Llama: Open foundation models for code," 2023, *arXiv:2308.12950*.

[47] H. Yu et al., "CodeReval: A benchmark of pragmatic code generation with generative pre-trained models," 2023, *arXiv:2302.00288*.

[48] M. Endres, S. Fakhoury, S. Chakraborty, and S. K. Lahiri, "Formalizing natural language intent into program specifications via large language models," 2023, *arXiv:2310.01831*.