

Enhancing Code Generation via Bidirectional Comment-Level Mutual Grounding

Yifeng Di, Tianyi Zhang
Purdue University, West Lafayette, USA
di5@purdue.edu, tianyi@purdue.edu

Abstract—Large Language Models (LLMs) have demonstrated unprecedented capability in code generation. However, LLM-generated code is still plagued with a wide range of functional errors, especially for complex programming tasks that LLMs have not seen before. Recent studies have shown that developers often struggle with inspecting and fixing incorrect code generated by LLMs, diminishing their productivity and trust in LLM-based code generation. Inspired by the mutual grounding theory in communication, we propose an interactive approach that leverages code comments as a medium for developers and LLMs to establish a shared understanding. Our approach facilitates iterative grounding by interleaving code generation, inline comment generation, and contextualized user feedback through editable comments to align generated code with developer intent. We evaluated our approach on two popular benchmarks and demonstrated that our approach significantly improved multiple state-of-the-art LLMs, e.g., 17.1% pass@1 improvement for code-davinci-002 on HumanEval. Furthermore, we conducted a user study with 12 participants in comparison to two baselines: (1) interacting with GitHub Copilot, and (2) interacting with a multi-step code generation paradigm called Multi-Turn Program Synthesis. Participants completed the given programming tasks 16.7% faster and with 10.5% improvement in task success rate when using our approach. Both results show that interactively refining code comments enables the collaborative establishment of mutual grounding, leading to more accurate code generation and higher developer confidence.

Index Terms—LLM, Code Generation, Code Refinement

I. INTRODUCTION

The quest for automated code generation, dating back to the 1960s [1], [2], has evolved significantly. This field has transitioned from early deductive program synthesis methods [3]–[6] to the recent advent of Large Language Models (LLMs) [7]–[9]. Despite the significant progress, LLMs often fail to align with developer intent due to factors such as reliance on spurious features, lack of user context, and misunderstanding of complex specifications [10]–[14]. Recent efforts to address these limitations include model fine-tuning [15], [16], new prompting strategies [17], [18], and iterative refinement paradigms [19], [20]. However, the improvement brought by these methods is still limited. For example, Self-Debug [19] only achieved a 4.8% pass@1 increase for Codex on MBPP when test execution feedback is not available.

Recent studies [11], [21], [22] highlight the critical role of bi-directional communication between developers and LLMs in programming tasks. While developers can edit prompts, LLMs often treat such edits as new prompts, hindering their ability to understand and incorporate developers’ refinement

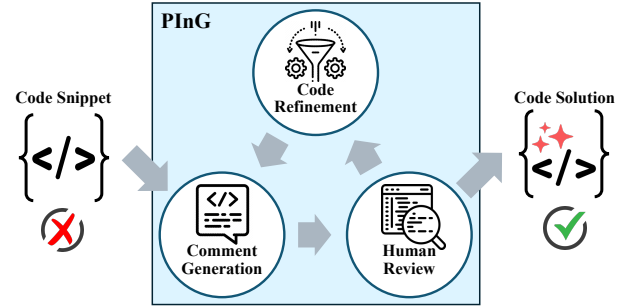


Fig. 1. Generating a code solution in our pipeline.

intent. Conversational models like ChatGPT offer multi-turn dialogues for feedback. However, effectively encoding historical utterances and contextualizing feedback within conversations remains a complex and challenging task [23]–[26].

In this work, we propose a new interactive approach called Programming with Interactive Grounding (PING). Figure 1 illustrates this approach. PING employs inline comments as a medium for bi-directional communication between a developer and the model. This approach is inspired by the grounding theory in communication [27], which underscores the importance of mutual understanding in collaborative interactions. Unlike prior work that relies on coarse-grained code explanations for refinement [19], [28], PING’s use of inline comments offers a more fine-grained approach. These comments directly address individual statements of the code, making it easier to target specific code segments for precise feedback and refinement.

Given a code snippet generated by an LLM, PING uses a code comment generation model to create inline comments that clarify each statement’s behavior. These comments provide developers with an immediate, understandable code description, helping them quickly spot potential errors. Developers can edit the comments to specify the correct behavior for the erroneous statement. A code refinement model then regenerates the statement and subsequent segment identified by the feedback, rather than the entire code snippet.

Inline comments are essentially natural language descriptions, which are suitable for developers at all levels. This approach allows for more precise error identification compared to prompt editing or conversational models, which often require the model to infer the location of errors throughout the entire code snippet. Comment editing also directly indicates

error locations, leading to more efficient model-based code refinement. This simplifies and improves the process of code refinement compared to previous approaches that require re-generating the entire code snippet.

We evaluate PING through simulated and real user studies on multiple code generation models. In our simulated user studies, we benchmark PING against eight state-of-the-art code generation and refinement techniques [19], [28]–[34], demonstrating that PING significantly outperforms these baselines. On HumanEval, pass@1 rates increased by 17.1% for code-davinci-002, 9.7% for InCoder, and 11.6% for CodeGen. MBPP benchmark also saw notable gains. In a user study with 12 real programmers, PING outperformed GitHub Copilot [35] and Multi-Turn Program Synthesis [36] in task success rate by 16.7% and 58.3%, respectively, while improving the task completion speed by 10.5% and 22.9%, respectively. Besides, participants reported 20% increased confidence and satisfaction, demonstrating PING’s effectiveness in enhancing code quality and user experience.

In summary, our main contributions include:

- 1) We propose PING, a new interactive code generation paradigm that enhances developer-model grounding via comment refinement.
- 2) We implement PING as a VSCode extension and have open-sourced our code and data.¹
- 3) We comprehensively evaluate PING with various models, benchmarks, and user studies, showing significant improvements in code accuracy and developer productivity over state-of-the-art methods.

II. BACKGROUND

Effective human interaction requires establishing *common ground*—mutual knowledge, beliefs, and assumptions [27]. Grounding in communication involves a collaborative process [37], where speakers design utterances for listener comprehension, and listeners provide feedback to demonstrate understanding [38]. Clark and Schaefer’s contribution model [37] outlines this as a presentation by the speaker and acceptance by the listener, supported by mechanisms including *repetition*, *reformulation*, and *elaboration* [39]. The importance of common ground is underscored when communication falters due to participants’ differing backgrounds [40].

Recent studies in code generation reveal challenges due to poor bi-directional communication between models and developers [11], [21], [22]. Models often misinterpret developer intent, while developers struggle to understand the generated code [11]. This communication gap, stemming from the lack of common ground, leads to code that does not align with developers’ expectations and hinders effective feedback [22]. Thus, fostering shared understanding between models and developers is a key to enhancing code generation accuracy.

In NLP, there have been some recent investigations on applying grounding theory to text-generation tasks [41], [42]. Zhang et al. [41] proposed a retrieval-augmented generation

model that retrieves related examples to ground the generative context in the relevant information. Chandu et al. [42] analyzed coordination and constraints in NLP tasks, proposing adaptations for grounding. However, grounding in code generation is still underexplored. Our work aims to fill this gap with a grounding-based pipeline for code generation and refinement, enhancing developer-model communication.

III. APPROACH

We formally define the task as follows: Given an initial context c that comprises a natural language problem description and an initial code snippet x produced by a code generation model, the task is to generate a refined code solution \hat{x} that fixes the errors in x and aligns the code more closely with the developer’s intent as articulated in the problem description.

We introduce Programming with Interactive Grounding (PING), which leverages inline code comments for bidirectional communication. PING includes three steps: comment generation, human review, and code refinement. We elaborate on each step in the following sections.

A. Comment Generation

PING first parsed the initial code snippet x into an Abstract Syntax Tree (AST) to segment the code snippet into individual statements. We opted for statement-level comment generation due to its fine-grained nature, which allows for a more accurate and detailed explanation of code behavior. This approach contrasts the previous approach that leverages coarser-grained code explanations or summaries, by providing targeted insights into specific statements of code.

We do not consider ‘import’ and ‘definition’ statements since they are trivial to explain. For compound statements such as ‘if’, ‘for’, ‘while’, ‘try’, and ‘with’, we adopt a tailored approach for code segmentation. For such nodes with a conditional statement, the condition expression is first added to the statement list, followed by its body statement nodes. The statements in the ‘else’ clause, if present, are handled afterward. For ‘try’ nodes, the process directly navigates through its body nodes without additional processing. For ‘with’ statements, the part of the statement excluding its ‘suite’ is first added, after which is its suite part.

If the generated code snippet x is not compilable, which prevents its transformation to AST, we resort to a simple strategy. In such cases, x is directly segmented into individual lines for subsequent comment generation.

Given individual statements, a specialized comment generation model M_C generates an inline comment c for each of them. Figure 2 ① shows a code example with generated comments. When processing the for loop, PING first generates an explanation for the loop condition. Then it proceeds to clarify the operations within the loop body.

To implement the comment generation model, we use CodeBERT [43] and fine-tune it with our finetuning dataset. CodeBERT is a transformer-based encoder-only model. It is pre-trained on a diverse corpus of programming languages and natural language text. To develop the fine-tuning dataset, we

¹<https://github.com/NecoraNyar/PInG>

Context c	1. Code Snippet with Comment \hat{x}	3. Regenerate based on the refined comment
<pre>def solve(N): """Given a positive integer N, return the total sum of its binary digits in binary. Example For N = 1000, the sum of digits will be 6 the output should be "110". For N = 150, the sum of digits will be 4 the output should be "100". For N = 147, the sum of digits will be 4 the output should be "100". Variables: @N integer Constraints: 0 ≤ N ≤ 10000. Output: a string of binary number """</pre>	<pre># Initialize the variable 'sum_of_digits' to 0 sum_of_digits = 0 # Convert N to a string digits = str(N) # Iterate through each digit in 'digits' for digit in digits : # Update 'sum_of_digits' by adding the integer value of 'digit' sum_of_digits += int(digit) # Convert the sum of digits to a binary, remove the first two characters and return return bin(sum_of_digits)[2:]</pre>	<pre># Initialize the variable 'sum_of_digits' to 0 sum_of_digits = 0 # Convert N to a binary and remove its prefix '0b' - digits = str(N) + digits = str(bin(N)).removeprefix('0b') # Iterate through each digit in 'binary_n' for digit in digits: # Update 'sum_of_digits' by adding the integer value of 'digit' sum_of_digits += int(digit) # Convert the sum of digits to a binary, remove the first two characters and return return bin(sum_of_digits)[2:]</pre>
<pre>Code Snippet x</pre> <pre>sum_of_digits = 0 digits = str(N) for digit in digits: sum_of_digits += int(digit) return bin(sum_of_digits)[2:]</pre>	<pre>2. Refine the comment of the incorrect line of code</pre> <pre>... - # Convert N to a string + # Convert N to a binary, and remove its prefix '0b' digits = str(N) ...</pre>	

Fig. 2. Refining a code snippet with our approach via comment editing.

adopted the approach outlined in the CodeBERT paper [43] and applied the preprocessing steps from CodeXGLUE [44] to the CodeSearchNet [45] dataset. The postprocessing process first cleans the data by (1) removing examples where the code cannot be parsed into an AST, (2) removing examples where the number of tokens in the document is fewer than 3 or greater than 256, (3) removing examples that contain special tokens (e.g., `` or `https://...`), and (4) removing examples that are not written in English. Then, it extracts comments and their corresponding code to produce a fine-tuning dataset with 280,652 code-to-comment pairs. We use Adam [46] to fine-tune the parameters. The learning rate was set to $5e^{-5}$, with a batch size of 32. We also set the maximum sequence length of input and inference as 256 and 128, respectively.

B. Human Review

In the human review step, the code snippet with inline comments is presented to the developer. When users notice an error by reading the comments, they can directly modify the corresponding comment to describe the desired behavior. For example, in Figure 2 ①, the developer notices the statement in Line 4 is wrong based on the comment of that statement. Specifically, instead of converting the input N to a string, it should convert it to a binary first. Fixing this bug requires changing this line of code to call three APIs: (1) converting the number to binary via `bin`, (2) transforming this binary to a string via `str`, and finally removing the string’s prefix via `str.removeprefix`. The developer may not be familiar with the first and third APIs as they are less commonly used. To manually fix it, the developer would typically need to search online and learn how to use these specific APIs first, which causes extra time and effort. Using PING, the developer can modify the comment instead, as shown in Figure 2 ②. Then, PING will regenerate the code based on the refined comment, which is more convenient for the developer.

C. Code Refinement

The original context c , the generated code up to the refined comment \hat{x} , and the refined comments are concatenated to form a new, updated context \hat{c} . A code refinement model, \hat{M} , then processes \hat{c} to regenerate code starting from the edited comment. This regeneration aims to address the identified error while maintaining the integrity of the correct parts.

To develop the code refinement model, we fine-tune the 6.7B version of DeepSeek Coder [47]. DeepSeek Coder is a recently released code generation model with superior coding performance while being small enough to be fine-tuned on our GPUs. Instead of using the original code generation model, we decided to fine-tune it, since the original one is designed to generate code based on general, high-level task descriptions. However, in our code refinement step, the task is to generate a dedicated program statement based on a specific, detailed inline comment. Therefore, fine-tuning is necessary.

To construct a high-quality dataset for code refinement, we utilize the Stack dataset [48], which includes 190.73 GB of Python code files. We first follow the filtering methods of Codex [7] and remove files with:

- an average line length greater than 100 characters
- a maximum line length above 1,000 characters
- less than 25% of the characters being alphanumeric
- keywords in the first few lines of the file indicating that the file was likely automatically generated

Then, we select code snippets that are sufficiently documented through comments. For each code file, we compute the comments-to-code ratio by counting the lines of comments and the lines of code. Finally, we set minimum and maximum thresholds for this ratio, selecting snippets within this range. After testing various thresholds, we established a MinRatio of 0.3 and a MaxRatio of 0.95 that optimally balances the number of code snippets against the richness of their comments.

We fine-tune the model with the next-token prediction objective, which is widely used in decoder-only models such as GPT-3. We use the Adam optimizer [46] with a learning rate

```

def solve(N):
    """
    Given a positive integer N, return the total sum of
    its binary digits in binary.
    Example
    For N = 1000, the sum of digits will be 6 the output
    should be "110".
    For N = 150, the sum of digits will be 4 the output
    should be "100".
    For N = 147, the sum of digits will be 4 the output
    should be "100".
    Variables: @N integer Constraints: 0 ≤ N ≤ 10000.
    Output: a string of binary number
    """
    # Initialize the variable 'sum_of_digits' to 0
    sum_of_digits = 0

    # Convert N to a binary, and remove its prefix '0b'
    digits = str(bin(N)).removeprefix('0b')
    # Iterate through each digit in 'binary_n'
    for digit in digits:
        # Update 'sum_of_digits' by adding the integer
        # value of 'digit'
        sum_of_digits += int(digit)

```

Fig. 3. User Interface of PING

of $2e^{-5}$ for updating the model weights. We use the Cross-Entropy [49] loss to optimize the model parameters. This loss compares the model's predicted probability distribution for the next token in a code or comment sequence against the ground-truth token. Mathematically, it is defined as:

$$\mathcal{L} = - \sum_{i=1}^N y_i \log(\hat{y}_i) \quad (1)$$

Here, N denotes the dictionary size, y_i is the ground-truth token, and \hat{y}_i is the model's predicted token. We compared the performance of our fine-tuned model with the original model. Please refer to Section IV for details.

D. Implementation Details

We have implemented PING as an extension for Visual Studio Code (VSCode). Figure 3 shows the user interface of this extension. The development of the PING extension for VSCode involved writing 4,605 lines of code. The default models used in PING are DeepSeek Coder for code generation, CodeBERT for comment generation, and the fine-tuned DeepSeek Coder model for code refinement.

IV. SIMULATED USER STUDY

In the evaluation, we strive to investigate to what extent PING improves the accuracy of different kinds of code generation models in a broad range of programming tasks under different settings. Since PING requires user feedback to guide the code refinement process, conducting such experiments at scale requires recruiting a large number of developers, which is costly and time-prohibitive to achieve in academia. Therefore, as a tradeoff, the first author acted as a simulated user and interacted with PING in the experiments in this section. In other words, the first author manually inspected

the code generated by PING in each setting and edited the comments to provide feedback for code refinement (detailed in Section IV-B). With the simulated user, we investigate the following research questions:

- RQ1: How effectively can our interactive approach improve the code generation accuracy of different LLMs?
- RQ2: How does our approach compare with other code generation approaches?
- RQ3: How sensitive is our approach to different code comment generation models?
- RQ4: To what extent can fine-tuning improve the code refinement accuracy?

In the end, we collected a very large interaction dataset, including 1224 code snippets generated by four different LLMs for 306 programming tasks, their inline comments generated by different comment generation models, comments edited by the simulated user, and code snippets generated by different code refinement models based on the edited comments. We have released this dataset to support the reproducibility of our experiments and also facilitate the development of new code refinement models for the research community.

A. Benchmarks

We used two popular code generation benchmarks in this evaluation. **HumanEval** [7] includes 164 hand-written Python programming tasks. **MBPP** [50] includes 974 crowd-sourced Python programming tasks. Since some tasks of MBPP have ambiguous task descriptions, the authors created a sanitized version with 427 tasks. For our evaluation, we randomly sampled 142 tasks from this sanitized version (about one-third) as analyzing all 427 tasks manually takes a lot of effort.

A recent study found that the original test cases from HumanEval do not sufficiently cover corner cases and thus cannot reliably assess the correctness of LLM-generated code [51]. The authors later released **HumanEval+** and **MBPP+**, which extend HumanEval and MBPP with additional test cases. In our evaluation, we also measured model performance on HumanEval+ and MBPP+.

B. User Feedback Collection

For the total of 306 programming tasks from HumanEval and MBPP, we experimented with four LLMs to generate the initial code snippets. We selected these models since they are well-known code generation models with different performance levels. We explain each of them below.

CodeGen [36] leverages multi-task learning on textual and programming language tasks, CodeGen can effectively generate syntactically valid code while capturing natural language meaning. For our evaluation, we used its 16B version.

InCoder [52] utilizes a retrieve-and-edit approach in code generation. Given a text description, it first retrieves relevant code snippets and then edits them to match the input. Its hybrid pointer-generator network allows both copying and generating tokens. For our evaluation, we used its 6B version.

Code-davinci-002 is OpenAI's GPT-3-based code generation model. It generalizes well across programming languages

without task-specific training. As a close-sourced model, its model size is undisclosed. We used the API provided by OpenAI to access this model and test it in our experiments.

DeepSeek Coder [47] is a new series of code language models that show superior coding performance. These models are pre-trained with a large project-level code corpus with both the next token prediction and the fill-in-the-middle objectives. For our evaluation, we used its 6.7B version.

For each task, we analyzed the top one program generated by each model. This resulted in 1,224 code snippets. 738 of them fail to pass the test cases. Table I shows the distribution of these 738 incorrect solutions across these models. Two annotators manually inspected these incorrect code solutions and edited the comments to refine the code. One of the annotators is the first author, a graduate student with over nine years of programming experience, including three years of industry experience. The other annotator, who is not the co-author of this paper, is a graduate student invited externally with five years of programming experience. We describe the detailed annotation procedure below.

The first step is to identify the buggy statements in each code snippet by examining the comments. The two annotators first had a 30-minute session to get familiar with the annotation task and go over 10 code snippets together to practice. Subsequently, they split the remaining 728 code snippets into halves and each independently inspected 364 code snippets. During the inspection and fault localization process, they discussed in case of uncertainty. After finishing their first round of inspection, they exchanged to check each other’s results. The initial agreement level between the two annotators in this step is 0.85 in terms of Cohen’s Kappa, indicating a perfect agreement level [53]. This agreement level makes sense since, in most cases, it is obvious which line of code is incorrect. The annotators then discussed and resolved all disagreements.

The second step is to edit the comment to reflect the expected behavior of a buggy statement for PING to refine the code. Like the previous step, they first practiced together on 10 snippets and then independently edited comments in half of the remaining snippets. Upon completion, they inspected the edited comments with each other and checked whether they agreed on the edited comments. The initial Cohen’s Kappa is 0.72, indicating a substantial agreement level [53]. Compared with the previous step, the agreement level is a bit lower since, in some cases, the annotators disagreed on the clarity and specificity of edited comments. For example, one annotator found another annotator’s comment edit ambiguous, too trivial, or even confusing. The annotators discussed each comment edit they initially disagreed on and came up with a final edit that both of them found appropriate.

They then ran PING to refine the code with the edited comment for each code snippet. If the refined code still failed the test cases, the two annotators repeated the previous steps to provide feedback to the refined code. Due to the enormous manual effort in this process, the annotators only provided up to three rounds of feedback to the 251 incorrect solutions from the HumanEval dataset.

TABLE I
COMPARISON OF CODE GENERATION MODELS EVALUATED

	Model Size	Pass@1		Incorrect Solution #
		HumanEval	MBPP	
INCODER	6B	16.5	19.7	262
CODEGEN-MONO	16B	29.9	35.2	212
CODE-DAVINCI-002	Undisclosed	46.3	58.5	165
DEEPSEEK-CODER	6.7B	74.4	73.2	99

After this annotation process, we logged all the generated code in each iteration and code comments before and after editing to form the interaction dataset described at the beginning of this section. We reported the performance gains of each round of feedback in Table IV.

C. Evaluation Metrics

Following the Codex paper [7], we measured each model’s performance using pass@ k . We set k to 1 and only considered the top 1 program generated by each model in our experiments.

D. Comparison Baselines

Since RQ1 only aims to measure the performance gain achieved on different LLMs, we compared the pass@1 of each model with and without the augmentation of PING. We also compared the pass@1 of each model after each feedback iteration and reported the improvement trend over iterations.

RQ2 compares PING with other code generation and refinement techniques. Thus, we selected eight state-of-the-art methods as the baselines. To ensure comparison consistency, we use GPT-3.5 with the default temperature (1.0) and top_p (1.0) as the base model for PING and all baselines in our experiments. For iterative methods such as Self-Edit [32] and Self-Debugging [19], we set the iteration upper bound to three, since the performance of these methods often saturates after two or three iterations based on the experiments in their papers. We describe each method below.

- **ReAct** [29]. ReAct prompts an LLM to generate both a reasoning trace and an action plan in an interleaved manner. This allows the model to perform dynamic reasoning to adjust the action plan.
- **ToT** [30]. ToT structures potential reasoning paths in a tree-like manner, which goes beyond the linear reasoning of traditional Chain-of-Thought (CoT) prompting to enable the exploration of multiple reasoning pathways.
- **RAP** [31]. RAP enhances LLM’s problem-solving by using them as both reasoning agents and world models to generate actionable plans and conduct complex reasoning, which predict environmental states and simulate action outcomes for more effective problem-solving.
- **Self-Edit** [32]. Self-Edit improves code generation accuracy by executing the generated code, analyzing results, and guiding the fault-aware code editor to correct errors in a generate-and-edit cycle.
- **Self-Planning** [28]. Self-Planning employs a two-phase (planning and implementation) approach in LLMs to

TABLE II
PASS@1 (%) ON THE HUMANEval AND HUMANEval+ DATASETS

	HumanEval				HumanEval+			
	Original	PING	PING w/o Finetuning		Original	PING	PING w/o Finetuning	
INCODER-6B	16.5	29.9	13.4 ↑	26.2	9.7 ↑	12.2	25.6	13.4 ↑
CODEGEN-MONO-16B	29.9	43.9	14.0 ↑	41.5	11.6 ↑	28.0	41.7	13.7 ↑
CODE-DAVINCI-002	46.3	63.4	17.1 ↑	63.4	17.1 ↑	42.1	59.1	17.0 ↑
DEEPSEEK-CODER-6.7B	74.4	79.9	5.5 ↑	78.7	4.3 ↑	70.7	76.2	5.5 ↑
							74.4	3.7 ↑

TABLE III
PASS@1 (%) ON THE MBPP AND MBPP+ DATASETS

	MBPP				MBPP+			
	Original	PING	PING w/o Finetuning		Original	PING	PING w/o Finetuning	
INCODER-6B	19.7	30.3	10.6 ↑	27.5	7.8 ↑	16.9	26.8	9.9 ↑
CODEGEN-MONO-16B	35.2	45.8	10.6 ↑	45.1	9.9 ↑	33.1	44.4	11.3 ↑
CODE-DAVINCI-002	58.5	69.0	10.5 ↑	69.0	10.5 ↑	50.7	63.4	12.7 ↑
DEEPSEEK-CODER-6.7B	73.2	78.2	5.0 ↑	76.8	3.6 ↑	64.1	69.7	5.6 ↑
							67.6	3.5 ↑

TABLE IV
PASS@1 RATES ACROSS MULTIPLE ITERATIONS ON HUMANEval

	Original	HumanEval			
		1 iteration	2 iterations	3 iterations	
INCODER-6B	16.5	26.2 9.7 ↑	31.1 4.9 ↑	34.1 3.0 ↑	
CODEGEN-MONO-16B	29.9	41.5 11.6 ↑	45.7 4.2 ↑	48.2 2.5 ↑	
CODE-DAVINCI-002	46.3	63.4 17.1 ↑	67.7 4.3 ↑	70.1 2.4 ↑	
DEEPSEEK-CODER-6.7B	74.4	78.7 4.3 ↑	81.7 3.0 ↑	83.5 1.8 ↑	

enhance LLMs’ understanding and handling of complex code generation tasks.

- **Self-Debugging** [19]. Self-Debugging empowers LLMs to perform rubber duck debugging on their own generated code via few-shot demonstrations, natural language code explanation, and execution analysis.
- **SCoT** [33]. SCoT improves traditional Chain-of-Thought (CoT) prompting by incorporating the program structures to obtain structured CoTs, which leads to more organized and efficient code generation.
- **CodeChain** [34]. CodeChain prompts LLMs to generate modular code and refine it via self-revisions. It boosts accuracy by extracting, clustering, and reusing code sub-modules, emulating expert programming practices.

For RQ3, we evaluated how PING performs with a different comment generation model, Seq2Seq [54]. Prior research has validated Seq2Seq’s effectiveness in generating inline comments [55]. We compared the pass@1 rates when using the proposed CodeBERT model and the Seq2Seq model.

V. EXPERIMENT RESULTS

A. RQ1: Effectiveness on Different Code Generation Models

Table II shows the pass@1 results after a single refinement iteration of PING across different LLMs on both HumanEval and its more rigorous version, HumanEval+. Similarly, Table III shows the pass@1 results after a single refinement iteration of PING on MBPP and its more rigorous version, MBPP+.

Results shown in Table II reveal significant improvements in code generation accuracy through just one iteration of refinement. For instance, the code-davinci-002 model shows a remarkable improvement on the HumanEval dataset, with the pass@1 rate improved from 46.3% to 63.4%. This enhancement highlights the significant influence of focused, iterative feedback in correcting model misconceptions.

Similar patterns of improvement were discernible across other models, including InCoder and CodeGen. Notably, InCoder’s performance on the HumanEval dataset ascended from 16.5% to 29.9%, while CodeGen’s accuracy experienced a boost from 35.2% to 45.8% on the MBPP benchmark. These improvements again confirm the effectiveness of incorporating human insights into the code generation process.

The overarching findings from our investigation affirm the critical role of human feedback in guiding LLMs toward synthesizing more functionally accurate programs. The consistent improvements across different models further validate the hypothesis that collaborative interaction is key to effectively addressing the challenges of code generation. This collaborative paradigm facilitates a model’s ability to dynamically adjust its outputs based on constructive feedback, thereby incrementally moving closer to generating error-free, functional code.

We also conducted experiments to explore how extending the interactive refinement process beyond a single iteration impacts the improvements in code generation. Table IV demonstrates a consistent pattern of improvement across all evaluated models, with each additional iteration yielding positive gains, albeit at a diminishing rate. For instance, the InCoder model shows a continuous gain from a baseline of 16.5% to 34.1% after three iterations, marking a total improvement of 17.6%. Similarly, the CodeGen model’s performance ascends progressively, culminating in a 48.2% pass@1 rate, while the code-davinci-002 model reaches a peak of 70.1%.

In summary, our analysis of both single and multiple iterations of interactive refinement reveals a clear trajectory of improvement in code generation accuracy. The gains were

TABLE V
COMPARISON OF PING WITH OTHER CODE GENERATION APPROACHES ON HUMANEval AND MBPP BENCHMARKS

	HumanEval		MBPP	
	Pass@1	Time (sec)	Pass@1	Time (sec)
ReAct [29]	56.9	50.8	66.9	42.6
ToT [30]	54.4	62.4	65.5	51.3
RAP [31]	63.1	59.6	70.4	50.9
Self-Edit [32]	62.2	40.1	56.3	35.7
Self-Planning [28]	65.2	37.8	58.5	34.5
Self-Debugging [19]	61.6	42.7	59.9	36.1
SCOT [33]	61.0	48.2	46.5	44.2
CodeChain [34]	62.8	64.5	59.2	58.7
PING	65.9	35.2	71.1	29.3

consistent across diverse model architectures, showing collaboration is vital for complex code generation. The findings highlight the efficacy of human feedback in guiding LLMs toward higher accuracy levels, affirming the value of interactive approaches in code generation.

Finding 1: PING significantly improves the code generation capability of four different LLMs on multiple benchmarks, demonstrating the effectiveness of comment-level code refinement.

B. RQ2: Comparison to Other Prompting Approaches

Table V shows the pass@1 results of different code generation and refinement methods on the HumanEval benchmark. Note that the results on PING in this table are only based on one iteration of human feedback. The results highlight the performance of our approach, PING, compared to other leading LLM-based approaches in code generation.

Table V demonstrates that PING not only performs competitively but also surpasses other approaches on both HumanEval and MBPP benchmarks. It proves to be a highly effective approach for addressing the complexities of code generation and refinement with pass@1 of 63.4 and 69.0, respectively.

On the HumanEval benchmark, PING’s leading score of 63.4 marks a notable advancement in code generation, exceeding RAP, the closest competitor. This performance illustrates PING’s effectiveness in producing correct code from natural language task descriptions. Similarly, on the MBPP benchmark, PING outperforms ReAct, demonstrating superior refinement capabilities by integrating human feedback. This success is largely due to PING’s innovative use of inline comments to align generated code with user intent for more precise code refinement.

Finding 2: PING outperformed state-of-the-art code generation and refinement methods, demonstrating its effectiveness in leveraging human feedback for more accurate and contextualized code refinement.

C. RQ3: Sensitivity to Code Comment Generation Models

Table VI compares the pass@1 rates of PING on different LLMs when CodeBERT vs. Seq2Seq as the code comment generation model. The results reveal an advantage in favor of

TABLE VI
PASS@1 WITH DIFFERENT COMMENT GENERATION MODELS ON THE HUMANEval DATASET

	Original	Seq2Seq	CodeBERT
INCODER-6B	16.5	23.8 7.3 ↑	26.2 9.7 ↑
CODEGEN-MONO-16B	29.9	39.0 9.1 ↑	41.5 11.6 ↑
CODE-DAVINCI-002	46.3	60.4 14.1 ↑	63.4 17.1 ↑

using CodeBERT over Seq2Seq across all LLMs. However, the impact of using different code comment generation models is not significant for all three LLMs—around 2% to 3% differences in pass@1. This implies that while using a better comment generation model can contribute to the refinement process, the advantage of PING is more pronounced by leveraging human feedback to refine code comments. Even with the suboptimal Seq2Seq model, the pass@1 improvements over the original LLMs are around 7% to 14%.

In addition to measuring the impact of the code comment model on pass@1, we further measured the accuracy of the CodeBERT model. Due to the lack of ground-truth comments, the same two annotators from Section IV-B sampled 385 code-comment pairs from the simulation dataset and categorized the comments into three accuracy levels: *fully accurate*, *largely accurate but with missing information*, and *contain wrong information*. This sample size is statistically significant with a 95% confidence level and a margin of error of 5%. The annotators followed a similar data annotation process as in Section IV-B to first have a 1-hour session to go over 25 code-comment pairs together. Then, they independently annotated half of the remaining 360 pairs and exchanged their annotations for validation. The initial agreement between the two annotators is 0.76 in terms of Cohen’s Kappa, indicating a substantial agreement level [53]. Then, they discussed the disagreements to achieve a consensus.

Among 385 code-comment pairs, the majority (72%) of comments generated by PING are fully correct. 12% of the comments are largely correct but with missing information, while 16% contain wrong information. Given that PING only regenerates the code for user-revised comments, missing or wrong information in other unchanged comments has little impact on the regenerated code. Yet we acknowledge that they may confuse or distract developers, since developers may spend extra time scrutinizing the code and comments to figure out whether the missing information is because of incorrect code or simply a comment generation error.

Finding 3: PING’s performance is not significantly impacted by the choice of comment generation model, illustrating its robustness and the primary value of iterative human feedback in enhancing code generation accuracy.

D. RQ4: The Impact of Finetuning

RQ4 investigates how fine-tuning the code refinement model affects the code generation accuracy of PING. Column PING and Column PING w/o Finetuning in Table II and III compare

the results of PING with a fine-tuned code refinement model against its performance without fine-tuning.

Both the InCoder and CodeGen models show notable improvements in pass@1 rates on both datasets with fine-tuning. InCoder’s performance increased by 3.7% on HumanEval and 2.8% on MBPP, while CodeGen saw a 2.4% gain on HumanEval. These results highlight the effectiveness of fine-tuning in enhancing code refinement accuracy and understanding of user intent.

Finding 4: Fine-tuning code generation models enhances code refinement accuracy, with our PING approach yielding marked improvements over baselines and fine-tuning offering additional performance boosts.

VI. USER STUDY

To evaluate the real-world utility and usability of our grounding-based approach, we utilize our integrated extension for Visual Studio Code to conduct a within-subjects user study with 12 participants, including both college students and professional programmers. This study aims to investigate the following research question:

- **RQ5:** How useful is our interactive approach to real programmers in practice?

A. Participants

We recruited 12 participants (10 males, 2 females) from a diverse background, comprising 2 graduate students, and 10 professional developers. They were recruited by emailing student mailing lists at an R1 university and reaching out to experienced software developers through our personal network. 2 had 1-2 years of Python experience (early intermediate), 5 had 3-5 years of experience (late intermediate), and 5 participants had over 5 years of Python experience (expert). This choice allows us to assess how well our grounding-based approach performs across a spectrum of Python expertise.

B. Task

Our task selection was inspired by the programming tasks used by Xu et al. [56] and Vaithilingam et al. [11]. We first categorized the TranX Developer Study tasks [56] and tasks from the DS-1000 benchmark [57] into different common programming task types. We then employed stratified random sampling from the two datasets. This process resulted in a pool of 6 code generation tasks with different types, including File I/O, OS, Web Scraping, Web Server & Client, Data Analysis, and Data Visualization. The ground-truth code for those tasks ranges from 15 LOC to 50 LOC.

C. Comparison Baselines

In addition to PING, we used GitHub Copilot [35] and Multi-Turn Program Synthesis (Multi-Turn for short from henceforth) [36] as comparison baselines. GitHub Copilot, powered by Codex, is an AI assistant that suggests code completions but lacks explicit repair feedback loops. Multi-Turn is a paradigm that decomposes code generation into steps,

where the model generates subprograms in response to natural language instructions. Here, users can only accept the generated code and provide further instructions for subsequent steps. The individually generated subprograms are concatenated into a single, complete program as the final generated program. GitHub Copilot offers continuous code generation without direct feedback mechanisms, while Multi-Turn enables task decomposition and multi-turn interaction without any chances for refinement. We used the VSCode extension of GitHub Copilot for our studies. For Multi-Turn, since it originally lacked a VSCode extension, we developed one with a similar user interface to PING’s to ensure a fair comparison.

D. Procedure

Our study employed a within-subjects design to directly compare PING with two comparison baselines. The whole study lasted about 100-115 minutes.

Participants first received an overview of the procedure, completed a consent form, and filled out a pre-study questionnaire that collected their background information. Then, each study was divided into three sessions, with each focusing on a specific code generation tool.

During these sessions, participants were asked to use the designated tool to tackle the assigned programming task within 20 minutes. To minimize learning effects, the assignment orders of both tasks and tools are counterbalanced across participants to ensure that each task-tool pair has the same number of trials. In each study, a participant was required to use all three different tools on three different tasks, with the ordering of the three task-tool pairs randomized.

Each session began with a tutorial video about the particular tool in focus. A subsequent 5-minute practice period allowed participants to get familiar with the tool before delving into the actual tasks. For each task, participants were asked to study its task description and use the tool to generate the code first. Participants could review the code and then use the tool’s interactive features to refine it until they deemed the code to be correct. Participants are allowed to adopt any approaches to verify the correctness of their current programs, including writing unit tests. If participants found a task too challenging, they could choose to skip it.

After each session, participants filled out a post-task questionnaire to assess confidence in their final code, perceived success rates, and five different cognitive load questions from NASA TLX [58]. After wrapping up all sessions, participants completed a final survey to compare their experiences of using these three tools. We record all the screen activities for playback to aid in discussing observed behaviors. All the systems logged interaction events, including code editing and comment editing, to analyze participants’ behavior patterns.

E. User Performance Results

As shown in Figure 4, 10 of 12 participants successfully solved the assigned programming task using PING within the given time. In comparison, 8 participants solved the given task with GitHub Copilot, and 3 with Multi-Turn. Compared to

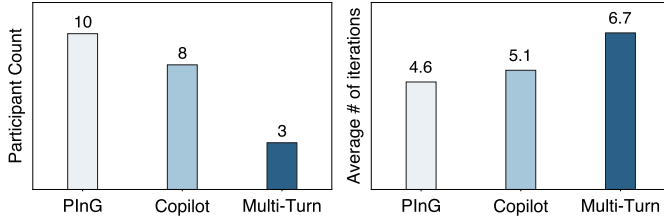


Fig. 4. The number of participants whose final code was functionally correct and the average number of iterations these participants took.

TABLE VII
THE AVERAGE TASK COMPLETION TIME (IN MINUTES)

Task Type	PInG	GitHub Copilot	Multi-Turn
File I/O	9.6	10.8	15.2
OS	13.7	16.2	18.8
Web Scraping	16.3	19.4	20
Web Server & Client	19.7	20	20
Data Analysis	11.5	13.6	20
Data Visualization	17.1	18.2	20
Average	14.65	16.37	19

GitHub Copilot and Multi-Turn, PING improved the success rate by 16.7% and 58.3%, respectively. It showcases the effectiveness of solving programming tasks by using inline comments to guide code generation.

Table VII compares task completion times across different tools. On average, participants finished their tasks 10.5% and 22.9% faster with PING when compared to GitHub Copilot and Multi-Turn, respectively. We further analyzed the average number of iterations participants tried to complete the given programming task with the assigned tool. Figure 4 shows these results. On average, participants using PING required fewer iterations (4.6) compared to GitHub Copilot (5.1) and Multi-Turn (6.7) to complete a programming task. This highlights how leveraging comments as a communication vehicle between users and LLMs improves the efficiency of identifying and resolving errors in LLM-generated code.

Although our study did not prohibit participants from writing unit tests, we observed that most participants chose not to write tests during the task. Instead, they relied on the inline comments generated by PING to inspect and understand code functionality quickly and effectively. At the end of each task, some participants opted to write one or two unit tests to double-check the correctness of their final code. This reflects a practical balance between immediate, comment-driven insights and traditional unit testing for comprehensive validation.

F. User Confidence and Cognitive Overhead Result

In the post-study survey, participants self-reported their confidence in solving the given programming task on a 7-point scale (1—very low confidence, 7—very high confidence). Notably, 9 of 12 participants agreed or strongly agreed that PING helped generate code that aligned with their intent, compared to 7 participants for GitHub Copilot and just 1 for Multi-Turn. Furthermore, PING significantly enhances

I felt confident about the code generated using the assigned tool.

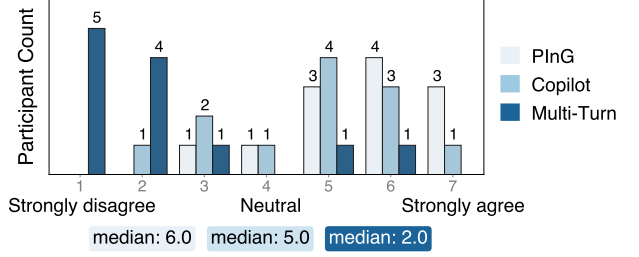


Fig. 5. The distribution of participants' confidence in the final code generated using the assigned tools.

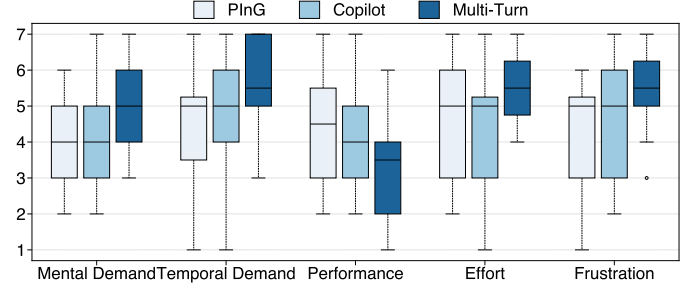


Fig. 6. The distribution of participants' cognitive load when solving the given programming tasks with the assigned tools.

developers' confidence in auto-generated patches. As shown in Figure 5, 7 participants agreed or strongly agreed that they felt confident about the generated code when using PING. In contrast, only 4 and 1 participant felt so when using GitHub Copilot and Multi-Turn, respectively. This can be largely attributed to PING's interactive refinement process, which empowers users to iteratively refine the generated code through inline comments, offering a more transparent and controlled development experience. This stark contrast in user confidence levels underscores the value of PING's approach, emphasizing the importance of a feedback loop in fostering trust and satisfaction with AI-generated code.

As shown in Figure 6, solving programming tasks with PING generally leads to a better user experience across all five cognitive load metrics compared to using Multi-Turn. Three of the five metrics show statistically significant differences between using PING and Multi-Turn based on the Wilcoxon signed-rank test (*Performance*: $p = 0.047$, *Effort*: $p = 0.038$, and *Frustration*: $p = 0.022$). This shows the benefits of using comments as the communication vehicle between users and LLMs. Compared to Copilot, PING slightly reduces the programming workload in terms of *Mental Demand*, *Temporal Demand*, *Performance*, and *Frustration*. However, these differences are not statistically significant.

Additionally, the user study shows our interactive approach assists real-world programming in multiple ways. For simple tasks, 67% of the participants directly used the code with minor edits. For complex tasks, while extensive changes were required, over 83% agreed the generated code gave useful

TABLE VIII
THE SUCCESS RATES ACROSS THREE DIFFICULTY LEVELS

	<i>PInG</i>	<i>GitHub Copilot</i>	<i>Multi-Turn</i>
Easy	4/4	4/4	2/4
Medium	4/4	3/4	1/4
Hard	2/4	1/4	0/4

structural scaffolds to build on. 75% of them reported that the inline comments enabled quickly identifying and fixing erroneous portions rather than debugging from scratch. On average, from 2-3 iterations, participants felt the code matched specifications sufficiently to serve as a quality starting point.

G. Impact of Task Difficulty Level

We categorized the programming tasks into three levels: easy, medium, and hard. Each level contains two tasks, which are classified based on the lines of code (LOC) in the ground truth code for each task. Tasks *File I/O* (16 LOC) and *Data Analysis* (22 LOC) are categorized as easy tasks. Tasks *OS* (27 LOC) and *Web Scraping* (37 LOC) are categorized as medium. Tasks *Web Server & Client* (46 LOC) and *Data Visualization* (51 LOC) are categorized as hard.

Table VIII shows the success rates for each difficulty level of programming tasks. We observe that participants using PING outperform those using the other two tools in medium and hard tasks. In easy tasks, PING and GitHub Copilot demonstrate equivalent success rates, both outperforming Multi-Turn. Table IX shows the average task completion time for each difficulty level. PING consistently saves time compared with GitHub Copilot and Multi-Turn across all difficulty levels.

H. Impact of User's Programming Expertise

The impact of programming expertise on solving programming tasks with code generation tools is noteworthy. Table X shows the success rates of participants with different levels of programming expertise and Table XI shows how participants' years of Python experience influenced their task completion time. Participants using PING perform better than those using the other two tools in hard tasks in terms of both success rates and average completion time. For easy and medium tasks, PING and GitHub Copilot show similar success rates, both significantly outperforming Multi-Turn. PING also saves much time over the other two tools in medium tasks. Surprisingly, we observed that programmers with 1-2 years of Python experience completed tasks faster using GitHub Copilot than with PING. This result suggests two potential reasons. One is that our tool might not be as beneficial for novices, who may struggle with reading code comments and understanding complex logic in natural language. The other one can be the small sample size of only two participants with 1-2 years of Python experience, which could make this result a coincidence. Overall, compared to programmers with less programming experience, those with longer programming experience solved the tasks faster across the three code generation tools, which is consistent with our intuition.

TABLE IX
THE AVERAGE TASK COMPLETION TIME FOR EACH DIFFICULTY LEVEL

	<i>PInG</i>	<i>GitHub Copilot</i>	<i>Multi-Turn</i>
Easy	10.5	12.2	17.6
Medium	15.0	17.8	19.4
Hard	18.4	19.1	20

TABLE X
THE SUCCESS RATES OF PARTICIPANTS WITH DIFFERENT LEVELS OF PROGRAMMING EXPERTISE

	<i>PInG</i>	<i>GitHub Copilot</i>	<i>Multi-Turn</i>
1-2 Years	1/2	1/2	0/2
2-5 Years	4/5	4/5	1/5
Over 5 Years	5/5	3/5	2/5

VII. LIMITATION AND FUTURE WORK

The results of our experiments demonstrate the potential for grounding-based interaction to significantly improve the accuracy and reliability of neural code generation models. Our approach, which leverages user feedback through inline comment editing, fosters a collaborative process of iterative alignment between the model's output and the user's intent. It underscores the synergy of human insights and model capabilities and thus makes code generation more reliable and adaptable through mutual grounding.

An interesting area for further analysis is studying the patterns in user edits that prove most effective for code refinement. Certain types of edits to comments, such as specifying additional conditions, correcting logical errors, or clarifying algorithm steps, may be particularly influential in guiding the model towards more accurate code refinement. Identifying and categorizing these high-leverage edits could inform techniques for eliciting more targeted feedback from users.

The comment generation and editing interface could also be enhanced to further optimize the grounding process. For example, highlighting model uncertainty and providing editing guidance could help users identify high-impact refinements more easily. Optimizing the cycle time between edits and regeneration could also affect overall productivity.

When applying PING to complex codebases, generating inline comments for each statement may make complex code look more overwhelming. This could hinder code readability, counter to clean code principles that emphasize simplicity and minimalism in annotations [59]. However, compared to simple code, complex code also benefits more from having detailed comments to facilitate program comprehension. We propose a couple of solutions to mitigate the negative effect of generating statement-level comments. First, we can allow users to hide all comments by default and only display the comments for the statements they do not understand. Second, we can develop a more advanced method that groups closely related statements to a block and only generates one comment for that block to reduce the number of comments.

TABLE XI
THE AVERAGE TASK COMPLETION TIME (IN MINUTES) OF PARTICIPANTS
WITH DIFFERENT LEVELS OF PROGRAMMING EXPERTISE

	<i>PlnG</i>	<i>GitHub Copilot</i>	<i>Multi-Turn</i>
1-2 Years	17.8	17.5	20
2-5 Years	14.7	16.1	19.5
Over 5 Years	13.3	16.2	18.1

In addition, our user study did not differentiate between participants who were previously familiar with the assigned programming tasks and those encountering them for the first time. Given that task familiarity could influence cognitive load and problem-solving strategies, as noted by Barke et al. [22], future work should incorporate this variable.

VIII. THREATS TO VALIDITY

A threat to internal validity is that our user feedback collected in the simulated user study was constructed by two experienced developers. The patterns in refinements and resulting accuracy improvements on this synthetic dataset may not fully reflect real-world usefulness. Our user study provides some mitigation by demonstrating productivity improvements with real users. However, the study was limited in scale and duration. More rigorous in-situ analysis is required to ascertain long-term productivity over continued tool usage.

Additionally, in our user study, we did not schedule breaks between tasks for participants. Executing tasks continuously without breaks may lead to fatigue among participants, potentially affecting their performance and the internal validity of our findings. Future studies should consider incorporating adequate breaks or spreading tasks across multiple sessions.

In terms of external validity, we only experimented with Python code generation, which may not generalize to other languages. Our set of models was also not exhaustive, so the benefits of grounding-based interaction for other model architectures are not fully characterized. The approach may be more or less effective for very small or very large models.

IX. RELATED WORK

In recent years, large language models have demonstrated unprecedented performance on code generation tasks [7], [9], [36], [47], [50], [52], [60], [61]. While these models share similar transformer architectures, they vary in training objectives, training datasets, and model sizes. For instance, Codex [7] is pre-trained on general text corpora like other GPT models and then fine-tuned on 54 million public software repositories hosted on GitHub. StarCoder [9] is pre-trained on a multi-lingual code corpus called Stack [48] and then fine-tuned on a Python-only code corpus. Furthermore, compared with Codex, which only uses next token prediction as the training objective, StarCoder also utilizes an additional training objective called fill-in-the-middle [62]. More recently, Magicoder [61] demonstrates that using LLMs to generate code instructions based on open-source code and then using them to fine-tune LLMs can significantly improve their code generation capability.

Among the various approaches of improving LLMs for code generation, the most related to us are the prompting methods for code generation and refinement [16], [19], [28]–[31], [31]–[34]. For instance, Self-Debugging [19] prompts LLMs to perform iterative debugging on the generated code based on test execution results and code explanations. However, unlike our approach, they generated high-level code explanations rather than inline code comments at the statement level. Our evaluation shows that inline comments can serve as a more effective method for grounding developer intent for code refinement, even when unit tests are not available.

In addition to test cases and code explanations, Nijkamp et al. [36] proposed a multi-step code generation paradigm where developers can express their intent step by step in a multi-turn dialogue with an LLM. Some approaches aim to automate this by using the same LLM for task decomposition or planning [28]–[31]. In these prompting paradigms, a developer or an agent needs to proactively decompose a task into smaller tasks to more precisely guide an LLM for code generation. By contrast, our work focuses on bi-directional communication where an agent explains the code to a developer, a developer pinpoints which part of the code is wrong, and the agent refines the erroneous part of the code accordingly.

Overall, our work differs from existing techniques by utilizing inline comments as a fine-grained grounding mechanism for code refinement. Furthermore, we have also developed specialized models for code comment generation and code refinement, rather than relying on the latent capabilities of LLMs for these specialized tasks. Our evaluation shows that developing such specialized models via fine-tuning is necessary given the distribution shift between the pre-training dataset and the targeted tasks.

X. CONCLUSION

In this work, we introduced an interactive pipeline to facilitate the grounding of code generation models to user intent through inline comment editing, which helps direct the code regeneration process to better match user expectations. Experiments with multiple code generation models on two popular benchmarks reveal notable improvements in code generation accuracy when performing grounding-based code refinement. Moreover, a user study shows productivity and usability benefits when compared to alternative code generation paradigms. Our approach enhances code generation to be more aligned with user intent, making the process more manageable. It highlights the synergy between human insights and models for improving code generation, suggesting future enhancements in human-AI collaboration research.

XI. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful feedback, as well as for the considerable time and effort they spent reviewing our work. We also thank the participants of the user studies for their valuable contributions and comments. This work was supported in part by NSF grants ITE-2333736 and CCF-2340408.

REFERENCES

- [1] R. J. Waldinger and R. C. Lee, "Prow: A step toward automatic program writing," in *Proceedings of the 1st international joint conference on Artificial intelligence*, 1969, pp. 241–252.
- [2] P. D. Summers, "A methodology for lisp program construction from examples," *Journal of the ACM (JACM)*, vol. 24, no. 1, pp. 161–175, 1977.
- [3] R. M. Burstall and J. Darlington, "A transformation system for developing recursive programs," *Journal of the ACM (JACM)*, vol. 24, no. 1, pp. 44–67, 1977.
- [4] Z. Manna and R. Waldinger, "Synthesis: dreams→ programs," *IEEE Transactions on Software Engineering*, no. 4, pp. 294–328, 1979.
- [5] —, "A deductive approach to program synthesis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 2, no. 1, pp. 90–121, 1980.
- [6] D. R. Smith, "Top-down synthesis of divide-and-conquer algorithms," *Artificial Intelligence*, vol. 27, no. 1, pp. 43–96, 1985.
- [7] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.
- [8] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.
- [9] R. Li, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, L. Jia, J. Chim, Q. Liu *et al.*, "Starcoder: may the source be with you!" *Transactions on Machine Learning Research*, 2023.
- [10] B. Kou, S. Chen, Z. Wang, L. Ma, and T. Zhang, "Do large language models pay similar attention like human programmers when generating code?" *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 2261–2284, 2024.
- [11] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Chi conference on human factors in computing systems extended abstracts*, 2022, pp. 1–7.
- [12] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, and T. Xie, "Codereval: A benchmark of pragmatic code generation with generative pre-trained models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.
- [13] Z. Z. Wang, A. Asai, X. V. Yu, F. F. Xu, Y. Xie, G. Neubig, and D. Fried, "Coderag-bench: Can retrieval augment code generation?" *arXiv preprint arXiv:2406.14497*, 2024.
- [14] Z. Wang, Z. Zhou, D. Song, Y. Huang, S. Chen, L. Ma, and T. Zhang, "Where do large language models fail when generating code?" *arXiv preprint arXiv:2406.08731*, 2024.
- [15] P. Haluptzok, M. Bowers, and A. T. Kalai, "Language models can teach themselves to program better," *arXiv preprint arXiv:2207.14502*, 2022.
- [16] A. Chen, J. Scheurer, J. A. Campos, T. Korbak, J. S. Chan, S. R. Bowman, K. Cho, and E. Perez, "Learning from natural language feedback," *Transactions on Machine Learning Research*, 2024. [Online]. Available: <https://openreview.net/forum?id=xo3hI5MwvU>
- [17] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, "No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 382–394.
- [18] Y. Choi and J.-H. Lee, "Codeprompt: Task-agnostic prefix tuning for program and language generation," in *Findings of the Association for Computational Linguistics: ACL 2023*, 2023, pp. 5282–5297.
- [19] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: <https://openreview.net/forum?id=KuPixIqPiq>
- [20] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via chatgpt," *arXiv preprint arXiv:2304.07590*, 2023.
- [21] C. Bird, D. Ford, T. Zimmermann, N. Forsgren, E. Kalliamvakou, T. Lowdermilk, and I. Gazit, "Taking flight with copilot: Early insights and opportunities of ai-powered pair-programming tools," *Queue*, vol. 20, no. 6, pp. 35–57, 2022.
- [22] S. Barke, M. B. James, and N. Polikarpova, "Grounded copilot: How programmers interact with code-generating models," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 85–111, 2023.
- [23] C.-W. Liu, R. Lowe, I. V. Serban, M. Noseworthy, L. Charlin, and J. Pineau, "How not to evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation," *arXiv preprint arXiv:1603.08023*, 2016.
- [24] Z. Tian, R. Yan, L. Mou, Y. Song, Y. Feng, and D. Zhao, "How to make context more useful? an empirical study on context-aware neural conversational models," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, 2017, pp. 231–236.
- [25] C. Sankar, S. Subramanian, C. Pal, S. Chandar, and Y. Bengio, "Do neural dialog systems use the conversation history effectively? an empirical study," *arXiv preprint arXiv:1906.01603*, 2019.
- [26] W. Zheng and K. Zhou, "Enhancing conversational dialogue models with grounded knowledge," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, 2019, pp. 709–718.
- [27] L. B. Resnick, J. M. Levine, and S. D. Teasley, *Perspectives on socially shared cognition*. American Psychological Association, 1991.
- [28] X. Jiang, Y. Dong, L. Wang, Q. Shang, and G. Li, "Self-planning code generation with large language model," *arXiv preprint arXiv:2303.06689*, 2023.
- [29] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafraan, K. R. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," in *The Eleventh International Conference on Learning Representations*, 2022.
- [30] S. Yao, D. Yu, J. Zhao, I. Shafraan, T. L. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," *arXiv preprint arXiv:2305.10601*, 2023.
- [31] W. Wang, Y. Wang, S. Joty, and S. C. Hoi, "Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 146–158.
- [32] K. Zhang, Z. Li, J. Li, G. Li, and Z. Jin, "Self-edit: Fault-aware code editor for code generation," *arXiv preprint arXiv:2305.04087*, 2023.
- [33] J. Li, G. Li, Y. Li, and Z. Jin, "Enabling programming thinking in large language models toward code generation," *arXiv preprint arXiv:2305.06599*, 2023.
- [34] H. Le, H. Chen, A. Saha, A. Gokul, D. Sahoo, and S. Joty, "Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules," *arXiv preprint arXiv:2310.08992*, 2023.
- [35] "GitHub Copilot · Your AI pair programmer," 2023. [Online]. Available: <https://github.com/features/copilot>
- [36] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.
- [37] H. H. Clark and E. F. Schaefer, "Contributing to discourse," *Cognitive science*, vol. 13, no. 2, pp. 259–294, 1989.
- [38] H. H. Clark and G. L. Murphy, "Audience design in meaning and reference," in *Advances in psychology*. Elsevier, 1982, vol. 9, pp. 287–299.
- [39] S. E. Brennan and H. H. Clark, "Conceptual pacts and lexical choice in conversation," *Journal of experimental psychology: Learning, memory, and cognition*, vol. 22, no. 6, p. 1482, 1996.
- [40] E. A. Isaacs and H. H. Clark, "Ostensible invitations1," *Language in society*, vol. 19, no. 4, pp. 493–509, 1990.
- [41] Y. Zhang, S. Sun, M. Galley, Y.-C. Chen, C. Brockett, X. Gao, J. Gao, J. Liu, and B. Dolan, "Dialogpt: Large-scale generative pre-training for conversational response generation," *arXiv preprint arXiv:1911.00536*, 2019.
- [42] K. R. Chandu, Y. Bisk, and A. W. Black, "Grounding 'grounding' in nlp," *arXiv preprint arXiv:2106.02192*, 2021.
- [43] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.
- [44] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

- [45] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.
- [46] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.
- [47] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming—the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.
- [48] D. Kocetkov, R. Li, L. Ben Allal, J. Li, C. Mou, C. Muñoz Ferrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf, D. Bahdanau, L. von Werra, and H. de Vries, "The stack: 3 tb of permissively licensed source code," *Preprint*, 2022.
- [49] R. Rubinstein, "The cross-entropy method for combinatorial and continuous optimization," *Methodology and computing in applied probability*, vol. 1, pp. 127–190, 1999.
- [50] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.
- [51] Y. Liu, C. Tantithamthavorn, Y. Liu, and L. Li, "On the reliability and explainability of automated code generation approaches," *arXiv preprint arXiv:2302.09587*, 2023.
- [52] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *arXiv preprint arXiv:2204.05999*, 2022.
- [53] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.
- [54] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, vol. 27, 2014.
- [55] Y. Huang, H. Guo, X. Ding, J. Shu, X. Chen, X. Luo, Z. Zheng, and X. Zhou, "A comparative study on method comment and inline comment," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 5, pp. 1–26, 2023.
- [56] F. F. Xu, B. Vasilescu, and G. Neubig, "In-ide code generation from natural language: Promise and challenges. corr. abs/2101.11149 (2021)," *arXiv preprint arXiv:2101.11149*, 2021.
- [57] Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, W.-t. Yih, D. Fried, S. Wang, and T. Yu, "Ds-1000: A natural and reliable benchmark for data science code generation," in *International Conference on Machine Learning*. PMLR, 2023, pp. 18 319–18 345.
- [58] S. G. Hart and L. E. Staveland, "Development of nasa-tlx (task load index): Results of empirical and theoretical research," in *Advances in psychology*. Elsevier, 1988, vol. 52, pp. 139–183.
- [59] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.
- [60] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "Wizardcoder: Empowering code large language models with evol-instruct," in *The Twelfth International Conference on Learning Representations*, 2023.
- [61] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Source code is all you need," *arXiv preprint arXiv:2312.02120*, 2023.
- [62] M. Bavarian, H. Jun, N. Tezak, J. Schulman, C. McLeavey, J. Tworek, and M. Chen, "Efficient training of language models to fill in the middle," *arXiv preprint arXiv:2207.14255*, 2022.