

CODEIF-BENCH: Evaluating Instruction-Following Capabilities of Large Language Models in Interactive Code Generation

Peiding Wang¹, Li Zhang¹, Fang Liu^{*1}, Lin Shi^{*2}, Minxiao Li¹, Bo Shen³, An Fu³

¹School of Computer Science and Engineering, State Key Laboratory of Complex & Critical Software Environment, Beihang University

²School of Software, Beihang University

³Huawei Cloud Computing Technologies Co., Ltd. China
{wangpeiding,fangliu,shilin}@buaa.edu.cn

Abstract

Large Language Models (LLMs) have demonstrated exceptional performance in code generation tasks and have become indispensable programming assistants for developers. However, existing code generation benchmarks primarily assess the functional correctness of code generated by LLMs in single-turn interactions. They offer limited insight into LLMs' abilities to generate code that strictly follows users' instructions in multi-turn interaction scenarios. In this paper, we introduce CODEIF-BENCH, a benchmark for evaluating the instruction-following capabilities of LLMs in interactive code generation. Specifically, CODEIF-BENCH incorporates nine types of verifiable instructions aligned with the real-world software development requirements, which can be independently and objectively validated through specified test cases, facilitating the evaluation of instruction-following capability in multi-turn interactions. In both *Static Conversation* and *Dynamic Conversation* settings, we evaluate the performance of 7 state-of-the-art LLMs and summarize the important factors influencing the instruction-following ability of LLMs in multi-turn interactions, as well as potential directions for improvement.

CCS Concepts

- Software and its engineering; • Computing methodologies → Artificial intelligence;

Keywords

Instruction Following, Benchmark, Interactive Code Generation, Large Language Models

1 Introduction

In recent years, the remarkable advancements in Large Language Models (LLMs) for code generation [25, 30, 41] have significantly enhanced developers' coding efficiency through an interactive and conversational paradigm [2, 13]. In practical software development environments, as shown in Figure 1, the collaboration between developers and LLMs typically necessitates an interactive code generation process. Throughout this interactive workflow, developers frequently enhance and refine their initial instructions by adding supplementary details to better align with their requirements. Specifically, through systematic code execution and rigorous review processes, developers often identify and provide follow-up

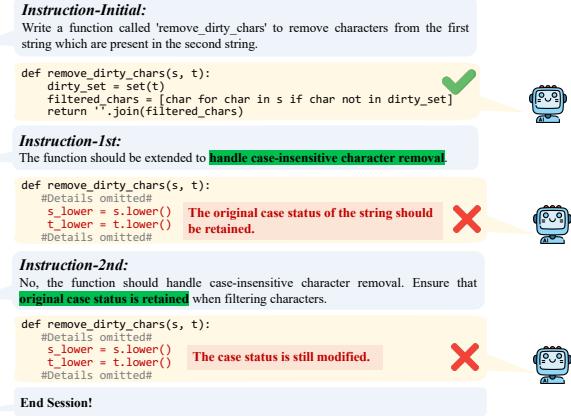


Figure 1: An example of interactive code generation, where the developer provides follow-up instructions to clarify the requirement and address issues in the generated code.

instructions to clarify requirements and address issues in LLM-generated code. These issues may include implementing new features [18], fixing bugs [32], resolving potential security vulnerabilities [28], etc. Specifically, any discrepancies or misunderstandings in LLM following instructions can significantly increase interaction time and cost, as developers may need to spend additional time clarifying and fixing errors. Besides, such misunderstandings might even trigger uncontrollable and unexpected issues, such as bugs [32], system failures [11], or even security vulnerabilities [28], which can have severe implications for the software's performance and developers' trust. Consequently, a dedicated benchmark to evaluate LLMs' **Instruction-Following (IF)** capabilities in interactive code generation is essential.

The key challenges for a benchmark to evaluate the instruction-following capabilities of LLMs in interactive code generation tasks include:

- **Challenge 1:** The benchmark data should support the multi-turn interaction with LLM in code generation.
- **Challenge 2:** The inherent complexity of natural language makes evaluation difficult.
- **Challenge 3:** The benchmark data should align with real-world development scenarios.

For **Challenge 1**, existing popular code generation benchmarks [3, 8, 15, 22] primarily evaluate whether LLMs can generate functionally correct code given specific programming tasks in a single turn, and it is difficult to directly evaluate LLMs' instruction-following capability in multi-round interaction. For **Challenge 2**, for instance, subjective instructions such as "Please make the comments clearer" may result in varying interpretations when assessing whether the model follows the given requirement. Some instruction-following benchmarks in the NLP domain, including for multi-turn [4, 14, 38], most use the LLM-as-Judge method for evaluation, which may lead to bias [7, 21]. To circumvent this challenge, some work, such as [40], focuses on evaluating the ability of LLMs to follow "verifiable instructions" which are defined as instructions that can be objectively verified for compliance, e.g., "the response should be more than 300 words". However, it does not address the **Challenge 3**: these benchmarks do not align with real-world software development scenarios and fail to evaluate LLM's ability to follow instructions in realistic interactive coding tasks.

To address the above challenges, we propose a new benchmark named CODEIF-BENCH, which aims to evaluate LLMs' instruction-following capability in interactive code generation. We first gather initial instructions from real-world projects [22] and crowd-sourced problems [3], enabling evaluation for both standalone function and repository-level code generation. Then we use a combination of LLM and manual annotation to extract nine primary verifiable instructions strategies of real-world user requirements to guide LLM to generate **verifiable instructions**, which can be independently and objectively verified using their respective test cases, such as initial instructions. We guide LLM to generate verifiable instructions based on these strategies and initial instructions, resulting in each data point in CODEIF-BENCH containing at least 8 verifiable instructions (8 for function-level and 10 for repository-level) to construct a conversation. In addition, we design *Static Conversation* (predefined instruction sequences) and *Dynamic Conversation* (providing feedback information), and evaluation metrics based on test cases to evaluate the performance of LLM in code interaction scenarios.

We evaluate the IF capabilities of 7 widely-used and advanced LLMs (*i.e.* GPT-4o [26], Claude-3.5-Sonnet [9], DeepSeek-V3 [41], Qwen-Coder-Turbo, Qwen2.5-Coder-{7B, 14B, 32B} [17]) in interactive code generation tasks using CODEIF-BENCH. Experimental results show the key factors, such as long context and the feedback utilization capability, influencing LLMs' instruction-following capabilities in interactive code generation tasks and potential methods for improvement. In summary, our contributions are as follows:

- To the best of our knowledge, we construct the first benchmark, CODEIF-BENCH, for evaluating the instruction-following capabilities of LLMs in interactive code generation.
- We present nine strategies derived from real software development to guide the verifiable instruction generation, which can be independently and objectively verified with their test cases.
- We design *Static Conversation* and *Dynamic Conversation* interactive scenarios, and quantitative metrics based on test cases to evaluate the instruction following capabilities of LLMs.

- We perform a thorough evaluation of 7 prominent LLMs on CODEIF-BENCH, analyzing the key factors influencing their IF capabilities in interactive code generation.

2 Related Work

Code Generation Benchmarks. Recent years have seen remarkable progress in LLM-based code generation, with models such as DeepSeek-Coder [41], WizardCoder [25], and Qwen-Coder [17] demonstrating exceptional performance. This progress has facilitated the development of interactive coding tools [2, 13] that enhance programmer productivity through effective human-LLM collaboration. This development has driven the development of benchmarks in code generation. These include benchmarks for evaluating LLMs' performance on various code generation scenarios, including stand-alone function generation (*e.g.*, HumanEval [8], MBPP [3], APPS [15]), class-level code generation (*e.g.*, ClassEval [10]), and repository-level code generation (*e.g.*, DevEval [22] and CoderEval [36], which leverage real-world open-source repositories). While these benchmarks primarily assess whether LLMs can generate functionally correct code in single-turn interactions, they are insufficient for evaluating LLMs' interactive capabilities, particularly in multi-turn interaction scenarios.

Instruction Following Benchmarks. Current benchmarks for evaluating the instruction-following capabilities of LLMs [20, 23, 38, 40] primarily focus on natural language QA tasks in the NLP domain. However, there is a significant disparity exists between code generation and QA tasks. Some of the existing benchmarks utilize LLM-as-judge to assess LLMs' instruction following performance [6, 16, 34, 38], which typically rely on elaborate prompts and can introduce inherent bias. CanItEdit [5] is a code editing benchmark, but its manually crafted programs lack project-specific dependencies, misaligning with real-world development. Additionally, its single-round dialogue design cannot assess LLMs' performance in complex, multi-turn interactions. CodeIF [35] benchmarks instruction-following in code generation but oversimplifies real-world complexity. Its single-turn setting also fails to assess LLMs' multi-turn interaction capabilities.

3 CodeIF-Bench

In this section, we first give a definition of the interactive code generation (§3.1) task and the evaluation metric (§3.2). Then we present the CODEIF-BENCH construction pipeline (§3.3).

3.1 Task Definition

In interactive code generation, users may refine their requirements through multiple rounds of dialogue, and LLMs generate code to meet users' expectations and make adjustments based on feedback. Specifically, for the N -th round, an LLM with strong instruction-following abilities can generate code that satisfies current instruction and the accumulated historical context from previous rounds.

For CODEIF-BENCH, every data contains multiple **verifiable instructions (VI)** accompanied by test cases. Each verifiable instruction can be independently validated through tests to determine whether the model's output follows the instruction. By linking instructions to their corresponding tests, CODEIF-BENCH evaluates the LLMs' instruction-following capabilities throughout the

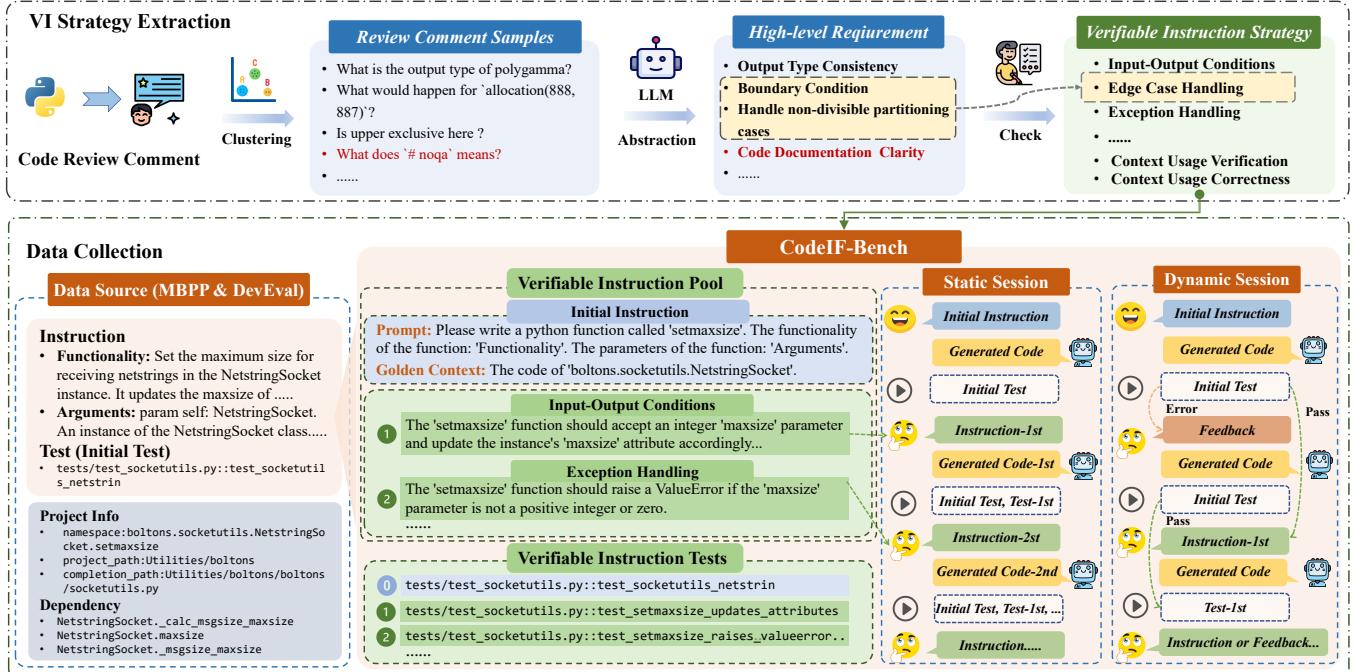


Figure 2: CODEIF-BENCH construction pipeline. The top part illustrates the verifiable instruction strategy extraction process, and the bottom part presents the data collection procedure.

interaction process by executing these tests. Specially, each data in CODEIF-BENCH is denoted as $D_i = \{(I_1, T_1), (I_2, T_2), \dots, (I_k, T_k)\}$, $i \in \{1, 2, \dots, |D|\}$, $k \in \{1, 2, \dots, |I|\}$, where $|D|$ is the total number of data and $|I|$ is the number of the verifiable instructions with tests. I_k represents the k -th verifiable instruction, and T_k represents the corresponding tests, which are both provided by the original benchmark or manual annotation. Based on CODEIF-BENCH, we define the following two classic interactive code generation scenarios:

① Static Conversation: In this scenario, we simulate a linear progression of requirements where instruction sequences are predetermined. The LLM generates code for each instruction in sequence without iterative refinement, testing its ability to maintain consistency in multi-turn interactions without external feedback. In an N -Round conversation, we initialize the process at $N = 1$ with the original programming instruction (I_1) from the benchmark. Subsequent instructions are then added sequentially from a predefined non-conflicting instruction sequence as N increases. In particular, for non-standalone tasks, which contain context-aware dependencies in repositories, I_1 consists of both the original task instruction and the golden contexts. Specifically, in i -th round, the input is formed as $Input = (I_1, A_1, I_2, A_2, \dots, I_i)$, LLM's answer in this round is A_i evaluated with the test cases (T_1, T_2, \dots, T_i).

② Dynamic Conversation: In real-world coding scenario, programmers typically review LLM-generated outputs to provide feedback. To simulate this human review process, we execute tests to obtain feedback and identify unmet requirements, thereby evaluating the LLM's IF capability in dynamic conversations. Specifically, we divide the entire session into two types of rounds: instruction rounds and feedback rounds. In each instruction round, instructions that did not meet the requirements in the previous round are selected through testing. If the instruction round fails, it enters

the feedback round for modification. To prevent the LLM from getting stuck in a loop, when both the instruction round and the corresponding feedback round fail, the instruction is discarded. Specifically, if A_i does not meet the desired requirements, it undergoes a feedback round, where I_{i+1} is constructed by incorporating I_i and the feedback F_i from the previous round. If A_i meets the desired requirements, or if it cannot meet the corresponding desired based on feedback, then it enters the instruction round: by executing tests, select the instructions that the A_i has not met.

3.2 Evaluation Metrics

In interactive code generation, we evaluate LLMs from three key perspectives. First, LLMs should not only follow the current instruction but also maintain consistent comprehension and execution across the entire dialogue. Second, we analyze the instruction forgetting phenomenon, where previously followed instructions are no longer respected in later turns. Third, in the *Dynamic Conversation* setting, where feedback is introduced, we assess instruction-following efficiency—i.e., how many interaction rounds are required to complete N instructions. Ideally, this would take exactly N rounds, but due to corrective feedback, it may require $2N$ or more.

Based on these perspectives, we propose and report the following evaluation metrics:

- **Instruction Accuracy (IA)** [14]: This metric measures the percentage of instructions that the LLM accurately follows in **each round** of conversation. Specifically, in the n -th round of dialogue, given the historical dialogue and the current instruction I_n , the LLM outputs A_n . Whether the LLM complies with instruction I_n

Table 1: Overview of the verifiable instruction strategies of CODEIF-BENCH. Two context-related strategies marked with * are customized for repository-level coding task.

Instruction Strategy	Description	L-1	L-2	L-3
Input-Output Conditions	Validity checking of input/output parameters that the code expects to receive	✓	✓	✓
Exception Handling	Mechanisms for catching and handling exception (error) conditions	✓	✓	✓
Edge Case Handling	Extreme boundary conditions for processing input data or operating scenarios	✓	✓	✓
Functionality Extension	Enhancement or extension of existing program functions	✓	✓	✓
Annotation Coverage	The extent to which annotations are applied in the code	✓	✓	✓
Code Complexity	Control the complexity of the code structure (e.g., Cyclomatic Complexity [12])	✓	✓	✓
Code Standard	Rules and guidelines for coding and structure (e.g., PEP 8 [33])	✓	✓	✓
Context Usage Verification*	Check and ensure that contextual information is used in the code	✗	✓	✓
Context Usage Correctness*	Check and ensure that contextual information is correctly used in the code	✗	✓	✓

is determined by judging whether A_n passes the test T_n .

$$IA = \begin{cases} 1, & \text{if } A_n \text{ passes the test } T_n \\ 0, & \text{if } A_n \text{ does not pass the test } T_n \end{cases} \quad (1)$$

- **Conversation Accuracy (CA):** This metric measures the percentage of instructions followed by LLMs from the first turn to the current turn in current turn. Specifically, in the n -th round of dialogue, given the historical dialogue $\{I_1, A_1, \dots, I_{n-1}, A_{n-1}\}$ and the current instruction I_n , the LLM outputs A_n . We calculate the CA score in this turn by executing the test sequence $TS = \{T_1, \dots, T_n\}$.

$$CA = \frac{\text{The Number of Tests Passed by } A_n \text{ in } TS}{\text{The Number of Tests in } TS} \quad (2)$$

- **Instruction Forgetting Ratio (IFR)** [14]: This metric assesses the percentage of instructions that are followed previously and not followed subsequently. Specifically, an instruction is considered forgotten if it was followed in one of the previous turns ($1, 2, \dots, n-1$) but is not followed in the current turn (n). The test sequence passed in the previous turn $PTS = \{T'_1, \dots, T'_k\}$ and the LLM outputs A_n .

$$IFR = \frac{\text{The Number of Tests Failed by } A_n \text{ in } PTS}{\text{The Number of Tests in } PTS} \quad (3)$$

- **Instructions Following Efficiency (IFE):** This metric evaluates the instruction following efficiency of LLM in **Dynamic Conversation**. It represents the average number of instructions followed per round from the initial round to the current round in the current round. Specifically, in the n -th round of dialogue, given the historical dialogue $\{I_1, A_1, \dots, I_m, A_m\}$ and the current instruction I_n , the LLM outputs A_n . We calculate the IFE score in this turn by executing the test sequence $TS = \{T_1, \dots, T_m, T_n\}$.

$$IFE = \frac{\text{The Number of Tests Passed by } A_n \text{ in } TS}{n} \quad (4)$$

3.3 Benchmark Construction

In this section, we present the benchmark construction pipeline. To align the data in CODEIF-BENCH with the distribution of real-world human instructions, as illustrated in Figure 2, our dataset construction process consists of two key steps: **VI Strategy Extraction** and **Data Collection**. In **VI Strategy Extraction**, we first extract verifiable instruction strategies from actual code review comments, identifying common patterns and deriving high-level rules aligning the distribution of real-world to guide the LLM in producing verifiable instructions be independently verified by tests.

In **Data Collection**, we use VI Strategies to guide LLMs generating verifiable instructions based on initial programming tasks, and the generated instructions undergo rigorous manual evaluation and quality assurance procedures to ensure their reliability and validity.

3.3.1 *VI Strategy Extraction.* During software development, human code reviewers systematically assess code quality across multiple dimensions, subsequently providing actionable improvement suggestions. Developers then iteratively refine their code based on the review comment, forming a collaborative process known as code review [24]. The code review process exhibits similarities to the interactive process between programmers and LLM-generated code. Inspired by this analogy, we leverage real-world code review comments [24] to extract instruction strategies, identifying common patterns for introducing follow-up requirements, aligning the distribution of human instructions in the real world. However, not all instruction types can be independently verified through test cases. As demonstrated in Figure 2, relatively subjective instructions (e.g., “What does # noqa mean?”) lack objective evaluation criteria, making them difficult to assess via automated testing. Furthermore, determining whether an instruction type can produce objective, verifiable outputs is non-trivial and often requires deep domain expertise. To address this, we employ manual annotation to filter verifiable instruction types serving as guidelines for generating diverse and actionable, verifiable instructions.

To extract diverse verifiable strategies, we first employ the KCenterGreedy [31] algorithm, which has demonstrated effectiveness in selecting a well-distributed set of representative samples based on text embeddings, to select 500 candidate review comment examples. (In our experiment, 500 is far greater than the actual number of unique instructions.) Then, we prompt GPT-4o to generate strategies for each candidate comment, which we refer to as “High-level Requirement” to guide LLM to generate verifiable instructions. To guarantee the validity and verifiability of these strategies, we invited 2 experienced software engineering experts with at least 4 years of programming experience to check these strategies, by removing unverifiable instruction types, merging similar ones, and refining ambiguous requirements into high-quality VI strategies. All ambiguous samples were resolved through discussion and mutual agreement to ensure strict adherence to the above criteria.

Finally, we obtain 9 verifiable instruction strategies, covering functional requirements (such as *Exception Handling*, *Input-Output Conditions*, etc) and non-functional requirements (such as *Annotation Coverage*, *Code Standard*, etc). Among these strategies, two are

Table 2: The statistics of CODEIF-BENCH. SA stands for StandAlone. Non-SA stands for repository-level code generation. Avg. L is the average lengths (tokens) of instructions calculated by GPT’s tokenizer.

Level	#Task	#Repo	Code Type		Dependency Type	#Total	Verifiable Instruction			Avg. L	
			SA (%)	Non-SA (%)			#Total	#Type	Test	Base	VI
L-1	70	14	100%	0%	SA	0	487	7	✓	43.8	17.2
L-2	40	26	0%	100%	intra_file	126	360	9	✓	9.6K	30.0
L-3	14	11	0%	100%	cross_file	98	126	9	✓	28K	30.0
Total	124	42	56.5%	43.5%	All	224	964	9	✓	7K	22.3

context-related: *i.e.*, *Context Usage Verification* and *Context Usage Correctness*, supporting the assessment of whether LLMs can effectively and accurately leverage the provided project-specific contexts for repository-level code generation tasks. Detailed information on VI strategies is illustrated in Table 1.

3.3.2 Data Collection. To construct CODEIF-BENCH, we first sample 124 programming tasks from two widely-used code generation benchmarks, where 50 tasks from MBPP [3] and 74 tasks from DevEval [22], encompassing both standalone functions and repository-level code generation. Specifically, for standalone function, we randomly sample 50 programming tasks from MBPP test dataset. The repo-level code generation tasks are sourced from DevEval, where the programming tasks are collected from 115 real-world projects across 10 domains covering different dependency types:

- **Level-1 (L-1):** Programming tasks that rely solely on built-in functions and standard libraries.
- **Level-2 (L-2):** Programming tasks that require intra-file or intra-class context.
- **Level-3 (L-3):** Programming tasks that require both intra-file and cross-file context.

We select data based on the 10 repository classes and dependency types provided by DevEval, randomly selecting 2 programming tasks from each class, resulting in 20 programming tasks for each dependency type. Due to the limited number of cross-file programming tasks, resulting in only 14 data points. Ultimately, we obtain 74 repository-level programming tasks. Then we generate the VIs for each task based on the extracted instruction strategies, and we also build corresponding unit tests (T) for assessing whether the code follows the VI. Inspired by Ou et al. [27], the instructions and tests are produced by LLMs, followed by human review and verification through execution. Specifically, the extracted VI strategies are applied to each task to guide the generation of verifiable instructions (7 for SA and 9 for Non-SA). Then we prompt GPT-4o to generate (VI, T) pairs based on the initial instruction and VI strategies. To further ensure data quality, we engaged two developers with over 4 years of Python programming experience to individually check and refine the data. They refined the dataset based on the following two criteria: 1. Ensuring *instruction description accuracy*—identifying vague descriptions or logical conflicts with the original programming task; 2. Validating *test correctness*—ensuring the test can verify instructions and introducing necessary dependencies to ensure proper execution of test functions. Through meticulous examination, they identified over 55% of the samples requiring refinement and achieved a Cohen’s Kappa score of 0.73. All ambiguous samples were resolved through discussion and mutual agreement to ensure strict adherence to the above criteria.

3.4 Dataset Overview

As shown in Table 2, we finally construct CODEIF-BENCH, which consists of 124 programming tasks along with 964 verifiable instructions. The tasks in CODEIF-BENCH can be categorized into 3 levels based on the scope of context usage (dependency). Table 3 shows a comparison between CODEIF-BENCH and existing benchmarks:

Code Domain & Real World Scenario. CODEIF-BENCH supports for both standalone function-level and repository-level code generation evaluation, where the programming tasks are derived from real-world projects [22] and crowd-sourced problems [3].

Robust Metric. CODEIF-BENCH enables an accurate evaluation of instruction following ability by calculating execution-based scores with corresponding test cases rather than LLM-as-Judge.

Multi-Round Support. CODEIF-BENCH enables the evaluation of LLMs’ ability to follow multi-turn instructions. Specifically, we constructed two types of dialogue scenarios—*Static Conversation* and *Dynamic Conversation* with feedback—to closely resemble real-world situations.

Table 3: Benchmark comparison.

Benchmark	Code Domain	Real World Scenario	Robust Metric	Multi-Round
Code Generation Benchmarks				
HumanEval [8]	✓	✗	✓	✗
MBPP [3]	✓	✗	✓	✗
CoderEval [36]	✓	✓	✓	✗
DevEval [22]	✓	✓	✓	✗
RepoEval [37]	✓	✓	✗	✗
CanItEdit [5]	✓	✗	✓	✗
NLP Instruction-Following Benchmarks				
MT-Bench [38]	✗	✗	✗	✓
Multi-IF [14]	✗	✗	✓	✓
InFo-Bench [29]	✗	✗	✗	✗
CODEIF [35]	✓	✗	✓	✗
CODEIF-BENCH	✓	✓	✓	✓

4 Evaluation

4.1 Research Questions

In this paper, we aim to answer the following research questions:

- **RQ1: Performance in Static Conversation.** How does LLM perform in instruction-following under *Static Conversation*?
- **RQ2: Performance in Dynamic Conversation.** How does LLM perform in instruction-following under *Dynamic Conversation*?
- **RQ3: Performance on Various VI Strategies.** How does LLM perform in instruction-following on different strategies of instructions?
- **RQ4: Exploration of IF Enhancement.** How do existing fine-tuning and prompt-enhancement methods perform in enhancing instruction-following capabilities in interactive coding tasks?

Table 4: IA results in Static Conversation. Values with darker colors indicate higher performance.

Level	Model	Turn-1	Turn-2	Turn-3	Turn-4	Turn-5	Turn-6	Turn-7	Turn-8	Turn-9	Turn-10	Avg.
L1	Claude-3.5-Sonnet	60.0	62.9	78.6	72.9	61.4	65.2	49.3	26.2	-	-	59.9
	GPT-4o	61.4	82.9	74.3	77.1	64.3	69.6	44.9	55.4	-	-	66.4
	DeepSeek-V3	52.9	74.3	74.3	67.1	45.7	68.1	37.7	55.4	-	-	59.5
	Qwen2.5-Coder-7B	45.7	68.6	71.4	60.0	38.6	65.2	33.3	41.5	-	-	53.2
	Qwen2.5-Coder-14B	48.6	70.0	70.0	60.0	38.6	65.2	30.4	43.1	-	-	53.3
	Qwen2.5-Coder-32B	51.4	75.7	75.7	75.7	47.1	68.1	39.1	46.2	-	-	60.0
	Qwen-Coder-Turbo	45.7	65.7	71.4	64.3	32.9	62.3	42.0	40.0	-	-	53.2
L2	Claude-3.5-Sonnet	60.0	45.0	50.0	52.5	40.0	52.5	10.0	55.0	32.5	27.5	42.5
	GPT-4o	35.0	35.0	40.0	37.5	12.5	17.5	5.0	20.0	15.0	12.5	23.0
	DeepSeek-V3	42.5	52.5	62.5	40.0	32.5	37.5	10.0	45.0	40.0	32.5	39.5
	Qwen2.5-Coder-7B	2.5	7.5	7.5	2.5	0.0	5.0	5.0	2.5	0.0	0.0	3.3
	Qwen2.5-Coder-14B	17.5	15.0	20.0	12.5	7.5	7.5	7.5	10.0	5.0	2.5	10.5
	Qwen2.5-Coder-32B	27.5	10.0	15.0	12.5	7.5	12.5	5.0	7.5	2.5	0.0	10.0
	Qwen-Coder-Turbo	35.0	17.5	30.0	17.5	5.0	10.0	7.5	5.0	7.5	5.0	14.0

4.2 Experimental Settings

As mentioned in Section 3.1, CODEIF-BENCH supports two scenarios of *N-Round* dialogue evaluation. For **Static Conversation**, the first round consists of the initial programming task, with verifiable instructions introduced in subsequent rounds. Since independent instructions could potentially conflict (e.g., ‘Functionality Extension’ instructions might interfere with existing code functionality), we predefined a set of non-conflicting instruction sequences to mitigate this issue (e.g., the ‘Functionality Extension’ instruction is added at the end of the conversation). In this paper, we report the experimental results based on the predefined instruction sequence.

For **Dynamic Conversation**, the initial programming task serves as the instruction of the first round, and then we execute the corresponding tests to determine whether feedback rounds are needed to improve the code. Next, run all the tests corresponding to the verifiable instructions to select the instruction that does not satisfy the current round, and continue to execute the tests to determine whether feedback is required for the round. The maximum number of rounds is set to twice the total number of instructions (16 for L1 and 18 for L2), as this is the maximum number of interactions in the worst-case scenario.

Due to limited budget and LLMs’ poor performance in L-3 (our preliminary experiments found that the most LLMs failed on L-3 tasks), only L-1 and L-2 tasks are included in the experiment. To ensure deterministic outputs for the evaluation, we employ the greedy decoding strategy, i.e., we set the parameters to “temperature=0”.

4.3 Evaluated LLMs

We select the following SOTA LLMs for code generation tasks:

- **Closed-Source LLM:** We chose Claude-3.5-Sonnet [9] and GPT-4o [1], powerful coding models in the field of code generation, and also introduced the closed-source models of the Qwen series, Qwen-Coder-Turbo [17].
- **Open-Source LLM:** We select one of the open-source large-scale SOTA LLMs Deepseek-V3 [41], and open-source small-scale SOTA code LLMs Qwen2.5-Coder-Instruct-7, 14, 32B [17].

5 Results and Analysis

5.1 RQ1: Performance in Static Conversation

To answer this research question, we evaluated the LLMs on CODEIF-BENCH under the *Static Conversation* setting. The **Instruction Accuracy (IA)** results are presented in Table 4. In the L-1 task, all LLMs exhibited strong IF capabilities, with IA averages exceeding 50%, and Qwen2.5-Coder-32B-Instruct even achieves performance comparable to closed-source models like Claude and larger open-source LLM DeepSeek-V3. In L-2 task, performance varied substantially across LLMs. For example, Claude completed 60% of tasks in Turn-1, whereas one of the best open-source models, Qwen2.5-Coder-32B-Instruct, achieved less than 30%. Moreover, as the dialogue rounds progressed, Claude and DeepSeek-V3 maintained relatively robust instruction-following performance, but GPT-4o exhibited a notable decline from the 5th round onward, while open-source models with fewer parameters (e.g., the Qwen series) degraded sharply after the 2nd round. We hypothesize that longer contexts disrupt weaker LLMs, hindering their ability to follow current instructions. Another key observation is that these LLMs exhibited performance declines in similar rounds, such as the 5th, 7th, and 8th rounds in L-1 data and the 7th round in L-2. As mentioned in Section 3.1, the instruction order is pre-determined in the static conversation setting. These rounds predominantly involved three instruction categories: ‘Annotation Coverage’, ‘Code Standard’, and ‘Functionality Extension’. The LLM exhibited notably poor adherence to instructions in these categories, and we further investigated this phenomenon in RQ3 (Section 5.3).

The **Conversation Accuracy (CA)** results are shown in Table 5. Since the instruction for the last round (Turn 8 in L-1, Turn 10 in L-2) is ‘Functionality Extension’, which conflicts logically with other instructions, i.e., there is no code that passes both the ‘Functionality Extension’ instruction and other instructions, we do not calculate the CA for Turn 8. On the L-1 dataset, all LLMs show varying degrees of performance degradation in CA as dialogue turns increase. Notably, nearly all models show substantial CA deterioration by the 7th round, and in L-2 (repository-level tasks), most models exhibit severe CA degradation starting from the second round, with near-complete breakdown in full-conversation instruction following by the 6th round. While most LLMs have relatively good ability

Table 5: CA results in Static Conversation. Values with darker colors indicate higher performance.

Level	Model	Turn-1	Turn-2	Turn-3	Turn-4	Turn-5	Turn-6	Turn-7	Turn-8	Turn-9	Avg.
L1	Claude-3.5-Sonnet	60.0	54.3	55.7	48.6	35.7	35.7	19.3	-	-	44.5
	GPT-4o	61.4	68.6	51.4	44.3	35.7	22.9	10.4	-	-	42.5
	DeepSeek-V3	52.9	58.6	47.1	41.4	22.9	15.7	3.0	-	-	34.8
	Qwen2.5-Coder-7B	45.7	57.1	48.6	35.7	15.7	11.4	4.5	-	-	31.5
	Qwen2.5-Coder-14B	48.6	52.9	41.4	32.9	12.9	8.6	1.5	-	-	28.6
	Qwen2.5-Coder-32B	51.4	64.3	52.9	47.1	22.9	17.1	8.9	-	-	38.1
	Qwen-Coder-Turbo	45.7	48.6	41.4	34.3	11.4	5.7	3.0	-	-	27.4
L2	Claude-3.5-Sonnet	60.0	37.5	30.0	25.0	10.0	5.0	0.0	0.0	0.0	18.6
	GPT-4o	35.0	17.5	12.5	10.0	2.5	0.0	0.0	0.0	0.0	8.6
	DeepSeek-V3	42.5	27.5	22.5	15.0	7.5	5.0	0.0	0.0	0.0	13.3
	Qwen2.5-Coder-7B	2.5	2.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.6
	Qwen2.5-Coder-14B	17.5	7.5	5.0	5.0	2.5	0.0	0.0	0.0	0.0	4.2
	Qwen2.5-Coder-32B	27.5	2.5	2.5	2.5	2.5	0.0	0.0	0.0	0.0	4.4
	Qwen-Coder-Turbo	30.0	5.0	2.5	2.5	2.5	0.0	0.0	0.0	0.0	4.7

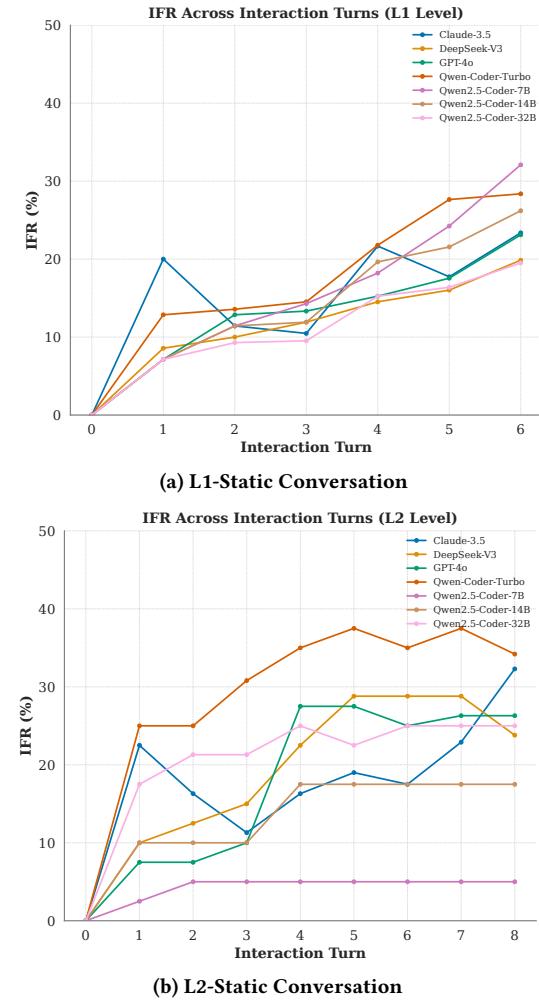
to follow current instructions (as demonstrated in the IA results), LLM’s performance in following instructions throughout the entire conversation is not satisfactory. We attribute this to long-context interference, including the context of the task itself and the context interference caused by the dialogue, which accelerates forgetting of prior instructions.

Figure 3 presents the **Instruction Forgetting Ratio (IFR)** results of LLMs. Most LLMs demonstrate a gradual increase in IFR as dialogue rounds progress, indicating that the frequency of forgetting phenomena rises with the accumulation of conversational instructions. Moreover, LLMs’ performance on L-2 data exhibits significantly higher IFR than on L-1 data, implying that the contextual complexity of L-2 (the necessary repository context for code completion) plays a crucial role in weakening LLMs’ ability to retain instruction states. Notably, certain models, such as Claude, display fluctuating IFR trends over successive dialogue rounds, suggesting that internal reasoning mechanisms may enable implicit corrective behaviors without explicit feedback.

RQ1 Summary: In *Static Conversation* without explicit external feedback, as the dialogue context expands, the LLM’s ability to follow current instructions (IA) and conversation instructions (CA) exhibits varying degrees of degradation influenced by model capabilities. While IA is further influenced by the instruction type, CA suffers from an additional decline due to instruction forgetting, where the Instruction Forgetting Rate (IFR) progressively increases as the conversation continues.

5.2 RQ2: Performance in Dynamic Conversation

Table 6 presents the **Instruction Accuracy (IA)** results on CODEIF-BENCH under the *Dynamic Conversation*. The introduction of feedback (compiler feedback and unfollowed instructions) significantly increases dialogue rounds. In the L-1 task, LLMs show different adaptability. Stronger models, such as Claude, demonstrate better robustness, and small-parameter open-source models gradually lose their ability to follow current-round instructions as the number of rounds increases. This indicates that stronger LLMs can effectively utilize external feedback to correct code and reduce the negative impact of the number of rounds for following current instructions.

**Figure 3: IFR results in Static Conversation.**

However, in the L-2 dataset, the IA performance of most LLMs shows varying degrees of decline. Notably, GPT-4o exhibits significant performance degradation compared to its performance in L-1.

Table 6: IA results in Dynamic Conversation. Values with darker colors indicate higher performance.

Level	Model	Turn 1-2	Turn 3-4	Turn 5-6	Turn 7-8	Turn 9-10	Turn 11-12	Turn 13-14	Turn 15-16	Turn 17-18	Avg.
L1	Claude-3.5-Sonnet	61.4	58.5	52.9	48.3	53.3	35.7	50.0	-	-	51.5
	GPT-4o	55.7	56.3	38.5	36.4	46.9	45.5	55.6	-	-	47.8
	DeepSeek-V3	51.4	54.5	50.9	36.2	35.3	50.0	70.0	-	-	49.8
	Qwen2.5-Coder-7B	35.0	34.8	16.1	17.0	5.7	3.7	0	-	-	16.0
	Qwen2.5-Coder-14B	40.0	36.2	19.5	13.9	20.8	21.1	18.2	-	-	24.2
	Qwen2.5-Coder-32B	46.4	39.9	25.0	12.1	11.5	9.5	8.8	-	-	21.9
	Qwen-Coder-Turbo	46.4	25.0	12.5	19.5	17.9	15.2	14.3	-	-	21.5
L2	Claude-3.5-Sonnet	53.8	32.5	21.1	31.0	30.4	43.8	35.7	33.3	57.1	37.6
	GPT-4o	16.2	8.8	10.3	2.8	0.0	0.0	0.0	0.0	0.0	4.2
	DeepSeek-V3	50.0	31.2	19.7	23.3	22.0	22.2	12.5	18.2	25.0	24.9
	Qwen2.5-Coder-7B	2.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3
	Qwen2.5-Coder-14B	12.5	6.2	6.2	2.6	0.0	5.3	0.0	2.9	0.0	4.0
	Qwen2.5-Coder-32B	13.8	3.8	3.8	0.0	1.3	0.0	1.4	0.0	2.8	3.0
	Qwen-Coder-Turbo	20.0	12.5	11.2	5.7	12.9	4.5	7.8	8.6	7.4	10.1

Table 7: CA and IFE results in Dynamic Conversation. Values with darker colors indicate higher performance.

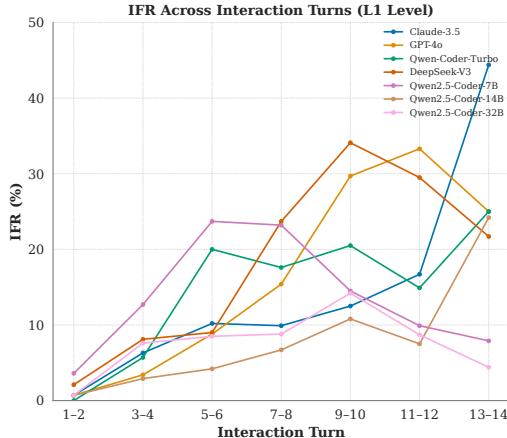
Level	Model	Turn 1-2		Turn 3-4		Turn 5-6		Turn 7-8		Turn 9-10		Turn 11-12		Turn 13-14		Turn 15-16		Turn 17-18		Avg. CA IFE	
		CA	IFE	CA	IFE	CA	IFE	CA	IFE	CA	IFE	CA	IFE	CA	IFE	CA	IFE	CA	IFE		
L1	Claude-3.5-Sonnet	63.6	56.1	73.5	57.9	70.0	50.9	68.5	46.0	61.7	40.9	58.1	37.8	51.0	32.9	-	-	-	-	63.8	46.1
	GPT-4o	60.7	58.2	69.9	55.4	62.0	45.2	59.7	41.4	45.4	31.2	32.7	24.0	36.6	27.9	-	-	-	-	52.4	40.5
	DeepSeek-V3	56.4	51.8	65.0	49.4	67.4	47.9	50.7	32.6	45.7	30.1	48.0	29.3	59.6	35.1	-	-	-	-	56.1	39.5
	Qwen2.5-Coder-7B	39.6	39.3	46.5	35.3	31.6	22.8	23.8	16.1	18.1	12.0	11.7	7.8	7.0	4.0	-	-	-	-	25.5	19.6
	Qwen2.5-Coder-14B	43.6	39.6	53.9	41.1	45.8	31.9	33.4	21.0	26.4	16.4	23.4	14.7	10.4	8.4	-	-	-	-	33.8	24.7
	Qwen2.5-Coder-32B	51.8	48.6	52.8	42.1	43.1	30.8	28.0	17.9	19.7	12.5	12.0	7.5	10.4	6.3	-	-	-	-	31.1	23.7
	Qwen-Coder-Turbo	51.4	47.5	48.2	36.6	28.2	18.9	26.2	16.4	24.2	15.4	21.6	13.4	8.9	5.9	-	-	-	-	29.8	22.0
L2	Claude-3.5-Sonnet	59.4	56.2	52.5	41.4	50.7	36.8	44.7	30.7	39.4	26.7	38.8	26.1	37.8	25.9	33.9	23.0	49.6	35.3	45.2	33.6
	GPT-4o	16.9	15.6	14.3	11.9	13.8	10.0	8.9	5.9	5.8	3.8	2.1	1.2	1.4	0.8	1.2	0.7	0.4	0.2	7.2	5.6
	DeepSeek-V3	54.4	51.2	49.5	38.7	44.7	32.2	40.3	26.4	38.9	24.6	34.0	20.7	30.6	18.1	18.3	10.9	19.5	11.8	36.7	26.1
	Qwen2.5-Coder-7B	3.8	3.8	1.0	0.7	0.7	0.5	0.6	0.3	0.5	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.7	0.6
	Qwen2.5-Coder-14B	13.8	13.8	6.8	5.2	7.0	5.0	2.4	1.5	0.9	0.5	7.8	4.4	5.6	3.0	2.8	1.5	0.0	0.0	5.2	3.9
	Qwen2.5-Coder-32B	15.0	13.8	7.0	5.6	5.5	4.0	2.4	1.6	0.7	0.4	0.8	0.5	1.0	0.6	0.9	0.5	1.4	0.8	3.9	3.1
	Qwen-Coder-Turbo	22.5	22.5	15.9	12.6	16.8	11.4	8.0	5.0	12.2	7.7	9.4	5.8	8.2	5.0	8.0	4.7	5.6	3.3	11.8	8.7

This indicates that the long context may prevent the LLM from effectively utilizing feedback and benefiting from it.

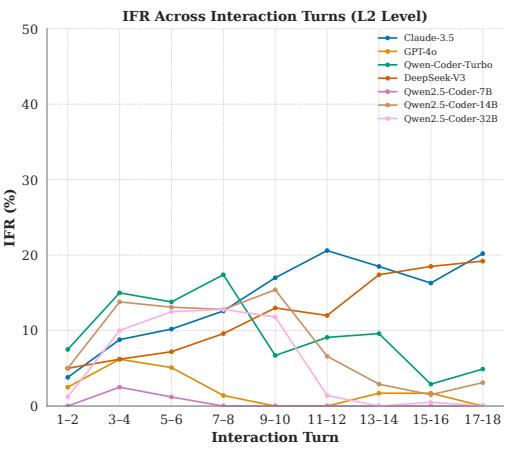
Table 7 presents the **Conversation Accuracy (CA)** results. Same as *Static Conversation*, we exclude “Functionality Extension” instructions in this experiment. In the L-1 task, compared to *Static Conversation* without feedback, LLMs like Claude and GPT4o showed substantial performance improvements (Comparison of Avg.: 44.5 vs 63.8 and 42.5 vs 52.4, respectively). However, the Qwen2.5-Coder series displayed only marginal improvements or even declines, revealing that the conversation performance gap between LLMs in dialogue tasks also stems from their differential ability to utilize feedback for code improvement. Interestingly, while DeepSeek-V3’s average CA score surpassed GPT4o’s, its IFE was lower, indicating that DeepSeek-V3 can effectively use feedback to enhance instruction compliance but requires more interaction rounds, thereby reducing interaction efficiency. This observation validates that our IFE and CA metrics provide a comprehensive evaluation of LLM performance. In L-2 data, Claude and DeepSeek-V3 outperformed other models, with CA metrics showing significant improvement over static dialogue settings, further demonstrating that improving models’ feedback utilization represents a promising direction for enhancing multi-round dialogue performance, especially when longer task context is required.

Additionally, we investigated the **Instruction Forgetting Rate (IFR)** of the LLMs. As shown in Figure 4, in the L1 task, the IFR of most LLMs gradually increases as the number of dialogue rounds grows, with Claude’s IFR surging to over 40% in rounds 13–14. In the L2 dataset, as dialogue rounds increase, Claude and DeepSeek-V3, which perform well in CA—show a gradually rising IFR, further suggesting that longer context length is a key factor in LLM forgetting. Notably, Qwen2.5-Coder-7B and 32B models exhibit a distinct pattern where IFR first rises and then declines. This behavior is because weaker LLMs are less capable of following new instructions, and in subsequent rounds, most of the instructions received are repeated prompts for previously correct but forgotten responses. This phenomenon mirrors real-world scenarios where users prompt LLMs to “recall” previous states, highlighting the critical role of user feedback and guidance in conversational performance.

Based on the above analysis, we identify that LLMs are more inclined to complete the current instruction and struggle to determine the scope of code edits. When faced with multiple instructions in extended conversations, LLMs fail to precisely delineate the editing boundaries for each instruction. This leads to this failure mode: when attempting to complete the current instruction, the LLM accidentally edits the previously correct part of the code, the model appears to “forget”. Consequently, Claude and DeepSeek-V3, which have stronger IA capabilities, exhibit higher IFR, thereby affecting



(a) L1-Dynamic Conversation



(b) L2-Dynamic Conversation

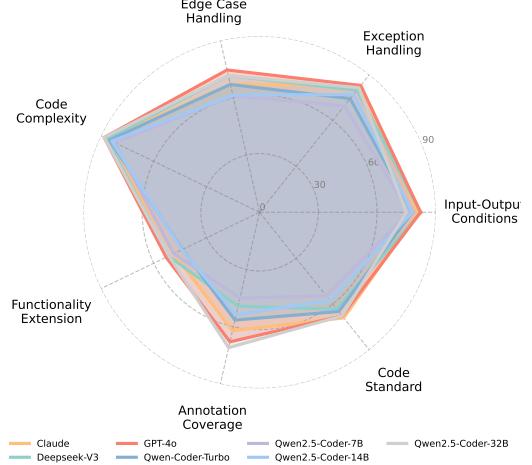
Figure 4: IFR results in Dynamic Conversation.

CA, while other LLMs with weaker IA capabilities received more reminders from users about forgetting them because they removed more instructions that they could not perform correctly, resulting in an initial increase followed by a decrease in IFR.

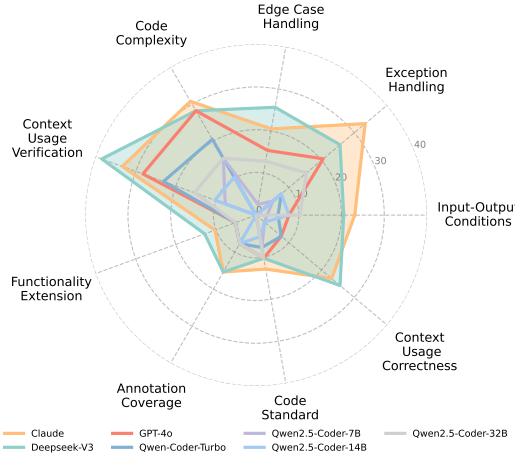
RQ2 Summary: In *Dynamic Conversation* with explicit external feedback, the LLM’s ability to effectively utilize feedback can substantially mitigate the negative effects of extended contexts on instruction following, particularly in long-context scenarios like repository-level code generation. Moreover, LLM struggles to determine the appropriate edit boundaries to accidentally edit the previously correct part of the code, which may result in instruction forgetting, ultimately degrading its instruction-following performance during interactions.

5.3 RQ3: Performance on Various VI Strategies

To evaluate LLMs’ instruction following performance across different instruction types, we implemented a 2-round experimental design: the first round comprised basic programming tasks, while the second round focused on verifiable instructions. This design effectively controlled for potential confounding effects from varying dialogue round counts. As illustrated in Figure 5, the L-1 dataset



(a) L1



(b) L2

Figure 5: The IA results on various VI strategies.

revealed significant performance variations in IA scores across instruction types, particularly in “Annotation Coverage”, “Code Standard”, and “Functionality Extension”. We also find that the underperforming Turns 5, 7, and 8 in Table 4 mostly correspond to these three instruction strategies. This suggests that LLMs may have insufficient training data for non-functional requirements (e.g., annotations and code style) and functional enhancement scenarios, resulting in comparatively weaker performance.

The results in the L-2 task further demonstrate substantial performance disparities between different LLMs and instruction types. Overall, the performance of LLM declined compared to L-1, further indicating that the long context of L-2 is an important factor affecting the instruction following ability of LLM. Closed-source models (e.g., Claude) and large-parameter open-source models (e.g., DeepSeek-V3) substantially outperformed small-parameter SOTA open-source models like Qwen2.5-Coder, particularly in development scenarios requiring longer context retention. Consistent with L-1 findings, instructions such as “Annotation Coverage” again showed relatively poor performance, reinforcing the observed limitations of LLMs in handling such requirements.

Table 8: IA and CA results across 8 interaction turns. Darker colors indicate higher performance.

Model	Turn-1		Turn-2		Turn-3		Turn-4		Turn-5		Turn-6		Turn-7		Turn-8		Avg.	
	IA	CA	IA	CA	IA	CA	IA	CA	IA	CA	IA	CA	IA	CA	IA	CA	IA	CA
Deepseek-Coder-6.7B	37.1	37.1	64.3	42.9	70.0	30.0	51.4	14.3	47.1	10.0	70.6	4.3	59.2	11.9	46.2	–	55.7	21.6
Deepseek-Coder-6.7B-CoT	34.3	34.3	61.4	41.4	71.4	24.3	55.7	20.0	47.1	7.1	63.6	1.4	54.7	13.4	44.7	–	54.1	20.4
OpenCodeInterpreter-DS-6.7B	32.9	32.9	58.6	40.0	64.3	32.9	44.3	20.0	40.0	11.4	61.7	7.1	47.3	14.9	39.3	–	48.6	22.8
CodeLlama-7B	35.7	27.1	47.1	30.0	64.3	15.7	52.9	8.6	44.3	4.3	83.8	0.0	64.8	0.0	20.1	–	50.2	12.4
CodeLlama-7B-CoT	34.3	22.9	12.9	7.1	20.0	0.0	15.7	2.9	25.7	0.0	21.7	0.0	10.2	0.0	8.2	–	17.3	4.8
OpenCodeInterpreter-CL-7B	30.0	30.1	22.9	10.0	67.1	20.0	27.1	5.7	27.1	1.4	17.4	0.0	25.9	1.4	15.3	–	28.1	9.9
CodeLlama-13B	30.0	30.0	38.6	25.7	44.3	12.9	45.7	5.7	50.0	7.1	52.2	0.0	40.5	0.0	23.3	–	40.7	11.8
CodeLlama-13B-CoT	21.4	21.4	17.1	7.1	55.7	7.1	44.3	0.0	35.7	0.0	44.5	0.0	56.0	0.0	12.7	–	36.1	5.2
OpenCodeInterpreter-CL-13B	22.9	22.9	37.1	22.9	62.9	18.6	30.0	8.6	41.4	7.1	20.2	5.7	47.4	6.0	27.6	–	36.3	13.2

Notably, LLMs exhibited strong performance in “Context Usage Verification”, indicating their capability to identify relevant contextual information while demonstrating poor performance in and “Context Usage Correctness”. This dichotomy suggests that while LLMs can effectively recognize correct contextual information, their ability to appropriately apply this context requires further enhancement.

RQ3 Summary: LLMs demonstrate suboptimal performance on instruction strategies such as “Annotation Coverage”, “Code Standard”, and “Functionality Extension”, likely due to insufficient training in these domains. Additionally, longer context lengths negatively impact the performance of LLM across all instruction types. While LLMs can effectively identify relevant information from context, their ability to utilize it accurately remains limited.

5.4 RQ4: Exploration of IF Enhancement

To investigate this question, we selected two representative methods: Chain-of-Thought [19] (CoT), a prompt engineering approach, and OpenCodeInterpreter [39], a fine-tuning-based method. For CoT, we adopted the original prompt template from its paper and applied it consistently across each dialogue round. OpenCodeInterpreter employs LLMs as simulators to interactively generate dialogue training data for code generation tasks, thereby enhancing the model’s performance in interactive coding scenarios. Given its substantial training dataset of 68K samples and generated by GPT-4, we chose it as the representative fine-tuning-based approach. For comparison, we evaluated the baseline model of OpenCodeInterpreter alongside DeepSeek-Coder-6.7B-Instruct, CodeLlama-7B-Instruct, and CodeLlama-13B-Instruct.

As shown in Table 8, CoT demonstrates a negative impact on improving LLMs’ instruction-following performance, with all metrics (including IA and CA) underperforming relative to the baseline. Our findings indicate that even when LLMs engage in structured, step-by-step reasoning, they frequently fail to produce correct outputs, and even the thought process is affected in its original ability to follow instructions, suggesting that easy prompt engineering alone is insufficient to enhance instruction following ability and may need carefully designed templates. Furthermore, while fine-tuning methods yield modest improvements in CA for DeepSeek-Coder-6.7B-Instruct and CodeLlama-13B-Instruct, the gains are marginal, and IA exhibits varying degrees of degradation. This may be because

OpenCodeInterpreter only uses LLM to generate data and does not further optimise data quality, thereby introducing noisy data caused by such as LLM hallucinations. This implies that although fine-tuning shows promise for improving LLM instruction-following in dialogue settings, further research is necessary to optimize its efficacy, particularly in areas such as data quality.

RQ4 Summary: Simply relying on thinking prompt templates (e.g., Chain-of-Thought) is insufficient to enhance an LLM’s instruction-following ability in conversational settings. And such reasoning processes may even degrade its baseline performance. While fine-tuning-based methods present a promising direction for improvement, further research is required to optimize their effectiveness.

6 Threats to Validity

Threats to External Validity arise from potential data leakage. While MBPP and DevEval are widely adopted benchmarks, they may have been exposed in public datasets used for model training. However, the verifiable instructions except for instructions from MBPP and DevEval in CODEIF-BENCH were meticulously manually annotated and not sourced from public corpora to mitigate leakage risks. Moreover, our experimental results demonstrate substantial performance variations across all evaluated LLMs on CODEIF-BENCH, with no observable fitting bias, suggesting that data leakage has minimal impact on evaluation accuracy. To further address this concern, we will implement periodic dataset updates to manage potential leakage risks proactively.

Threats to Internal Validity stem from the fact that CODEIF-BENCH is a single-language dataset focused solely on Python, neglecting other languages. However, our method is language-agnostic, and in future work, we will incorporate other programming languages such as Java. Additionally, the validity threat also arises from the strategies of instruction. Although we collected them from real review comments, they may not cover all instruction strategies. In future work, we will regularly update the instruction strategies and data to address this threat.

Threats to Construct Validity relate to manual annotation. Although we hired experienced programmers for annotation and reached consensus on all data through discussion, potential defects and errors may still exist. We have open-sourced the data to collect any errors that may arise and update our dataset on time.

7 Conclusion

This paper presents CODEIF-BENCH, the first benchmark for evaluating LLMs' instruction-following capabilities in interactive code generation. CODEIF-BENCH incorporates verifiable instructions aligned with the real-world software development, complemented by corresponding test cases, enabling a comprehensive evaluation of LLMs' instruction-following performance in multi-turn code generation scenarios. In both *Static Conversation* and *Dynamic Conversation* settings, we evaluate the performance of 7 state-of-the-art LLMs and summarize important influencing factors and possible areas for improvement. In our future research, we will expand CODEIF-BENCH by incorporating additional programming languages and exploring methods to enhance the interactive code generation capabilities of LLMs.

References

- [1] 2024. GPT-4o System Card. arXiv:2410.21276 [cs.CL] <https://arxiv.org/abs/2410.21276>
- [2] Anysphere. 2023. Cursor. <https://www.cursor.com/>
- [3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).
- [4] Ge Bai, Jie Liu, Xingyu Bu, Yancheng He, Jiaheng Liu, Zhanhui Zhou, Zhuoran Lin, Wenbo Su, Tiezheng Ge, Bo Zheng, and Wanli Ouyang. 2024. MT-Bench-101: A Fine-Grained Benchmark for Evaluating Large Language Models in Multi-Turn Dialogues. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 7421–7454. doi:10.18653/v1/2024.acl-long.401
- [5] Federico Cassano, Luisa Li, Akul Sethi, Noah Shinn, Abby Brennan-Jones, Jacob Ginesin, Edward Berman, George Chakhnashvili, Anton Lozhkov, Carolyn Jane Anderson, et al. 2023. Can it edit? evaluating the ability of large language models to follow code editing instructions. *arXiv preprint arXiv:2312.12450* (2023).
- [6] Dongping Chen, Ruoxi Chen, Shilin Zhang, Yinuo Liu, Yaochen Wang, Huichi Zhou, Qihui Zhang, Yao Wan, Pan Zhou, and Lichao Sun. 2024. Mlm-as-a-judge: Assessing multimodal llm-as-a-judge with vision-language benchmark. *arXiv preprint arXiv:2402.04788* (2024).
- [7] Guiming Hardy Chen, Shunian Chen, Ziche Liu, Feng Jiang, and Benyou Wang. 2024. Humans or LLMs as the Judge? A Study on Judgement Bias. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 8301–8327. doi:10.18653/v1/2024.emnlp-main.474
- [8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [9] claude. 2023. Claude. <https://claude.ai/>
- [10] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xia Peng, and Yiling Lou. 2023. ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-level Code Generation. *CoRR* abs/2308.01861 (2023). arXiv:2308.01861 doi:10.48550/arXiv.2308.01861
- [11] Murray Dunne, Kylene Schram, and Sebastian Fischmeister. 2024. Weaknesses in LLM-Generated Code for Embedded Systems Networking. In *2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 250–261.
- [12] Christof Ebert, James Cain, Giuliano Antoniol, Steve Counsell, and Phillip Laplante. 2016. Cyclomatic complexity. *IEEE software* 33, 6 (2016), 27–29.
- [13] Github. 2021. Github Copilot. <https://github.com/features/copilot>
- [14] Yun He, Di Jin, Chaoqi Wang, Chloe Bi, Karishma Mandyam, Heji Zhang, Chen Zhu, Ning Li, Tengyu Xu, Hongjiang Lv, Shruti Bhosale, Chenguang Zhu, Karthik Abinav Sankararaman, Eryk Helenowski, Melanie Kambadur, Aditya Tayade, Hao Ma, Han Fang, and Sinong Wang. 2024. Multi-IF: Benchmarking LLMs on Multi-Turn and Multilingual Instructions Following. arXiv:2410.15553 [cs.CL] <https://arxiv.org/abs/2410.15553>
- [15] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, Joaquin Vanschoren and Sai-Kit Yeung (Eds.). <https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c24cd76e1ce41366a4bbe8a49b02a028-Abstract-round2.html>
- [16] Hui Huang, Yingqi Qu, Jing Liu, Muyun Yang, and Tiejun Zhao. 2024. An empirical study of llm-as-a-judge for llm evaluation: Fine-tuned judge models are task-specific classifiers. *arXiv preprint arXiv:2403.02839* (2024).
- [17] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaior Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-Coder Technical Report. arXiv:2409.12186 [cs.CL] <https://arxiv.org/abs/2409.12186>
- [18] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyi Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770* (2023).
- [19] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2023. Large Language Models are Zero-Shot Reasoners. arXiv:2205.11916 [cs.CL] <https://arxiv.org/abs/2205.11916>
- [20] Wai-Chung Kwan, Xingshan Zeng, Yuxin Jiang, Yufei Wang, Liangyou Li, Lifeng Shang, Xin Jiang, Qun Liu, and Kam-Fai Wong. 2024. MT-Eval: A Multi-Turn Capabilities Evaluation Benchmark for Large Language Models. *arXiv preprint arXiv:2401.16745* (2024).
- [21] Dawei Li, Renliang Sun, Yue Huang, Ming Zhong, Bohan Jiang, Jiawei Han, Xiangliang Zhang, Wei Wang, and Huan Liu. 2025. Preference Leakage: A Contamination Problem in LLM-as-a-judge. *arXiv:2502.01534* [cs.LG] <https://arxiv.org/abs/2502.01534>
- [22] Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, Jiazhen Ding, Xuanning Zhang, Yuqi Zhu, Yihong Dong, Zhi Jin, Binhu Li, Fei Huang, Yongbin Li, Bin Gu, and Mengfei Yang. 2024. DevEval: A Manually-Annotated Code Generation Benchmark Aligned with Real-World Code Repositories. In *ACL (Findings)*. Association for Computational Linguistics, 3603–3614.
- [23] Youquan Li, Miao Zheng, Fan Yang, Guosheng Dong, Bin Cui, Weipeng Chen, Zenan Zhou, and Wentao Zhang. 2024. FB-Bench: A Fine-Grained Multi-Task Benchmark for Evaluating LLMs' Responsiveness to Human Feedback. arXiv:2410.09412 [cs.CL] <https://arxiv.org/abs/2410.09412>
- [24] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022. Automating Code Review Activities by Large-Scale Pre-training. arXiv:2203.09095 [cs.SE] <https://arxiv.org/abs/2203.09095>
- [25] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568* (2023).
- [26] Josh Achiam OpenAI, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2024. Gpt-4 technical report, 2024. URL <https://arxiv.org/abs/2303.08774> (2024), 6.
- [27] Jiao Ou, Jiayu Wu, Che Liu, Fuzheng Zhang, Di Zhang, and Kun Gai. 2024. Inductive-Deductive Strategy Reuse for Multi-Turn Instructional Dialogues. *arXiv preprint arXiv:2404.11095* (2024).
- [28] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2023. Do users write more insecure code with AI assistants?. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2785–2799.
- [29] Yiwei Qin, Kaiqiang Song, Yebowen Hu, Wenlin Yao, Sangwoo Cho, Xiaoyang Wang, Xuansheng Wu, Fei Liu, Pengfei Liu, and Dong Yu. 2024. InFoBench: Evaluating Instruction Following Ability in Large Language Models. arXiv:2401.03601 [cs.CL] <https://arxiv.org/abs/2401.03601>
- [30] Baptiste Roziere, Jonas Gehring, Fabian Gloclekle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
- [31] Ozan Sener and Silvio Savarese. 2018. Active Learning for Convolutional Neural Networks: A Core-Set Approach. In *International Conference on Learning Representations*.
- [32] Florian Tambon, Arghavan Moradi Dakhel, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Giuliano Antoniol. 2024. Bugs in Large Language Models Generated Code: An Empirical Study. arXiv:2403.08937 [cs.SE] <https://arxiv.org/abs/2403.08937>
- [33] Guido Van Rossum, Barry Warsaw, and Nick Coghlan. 2001. PEP 8-style guide for python code. *Python. org* 1565 (2001), 28.
- [34] Hui Wei, Shenghua He, Tian Xia, Andy Wong, Jingyang Lin, and Mei Han. 2024. Systematic evaluation of llm-as-a-judge in llm alignment tasks: Explainable metrics and diverse prompt templates. *arXiv preprint arXiv:2408.13006* (2024).
- [35] Kaiwen Yan, Hongcheng Guo, Xuanqing Shi, Jingyi Xu, Yaonan Gu, and Zhoujun Li. 2025. CodeIF: Benchmarking the Instruction-Following Capabilities of Large

- Language Models for Code Generation. arXiv:2502.19166 [cs.SE] <https://arxiv.org/abs/2502.19166>
- [36] Hao Yu, Bo Shen, Dezheng Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.
- [37] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570* (2023).
- [38] Liannmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhiuhuan Li, Dacheng Li, Eric Xing, et al. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems* 36 (2023), 46595–46623.
- [39] Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhui Chen, and Xiang Yue. 2024. OpenCodeInterpreter: Integrating Code Generation with Execution and Refinement. In *Findings of the Association for Computational Linguistics: ACL 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 12834–12859. doi:10.18653/v1/2024.findings-acl.762
- [40] Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. 2023. Instruction-following evaluation for large language models. *arXiv preprint arXiv:2311.07911* (2023).
- [41] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. *arXiv preprint arXiv:2406.11931* (2024).