DialogAgent: An Auto-engagement Agent for Code Question Answering Data Production

Xiaoyun Liang
ByteDance
Shenzhen, China
liangxiaoyun.duo@bytedance.com

Jingyi Ren
ByteDance
Shenzhen, China
jingyi.422@bytedance.com

Jiayi Qi
ByteDance
Beijing, China
qijiayi@bytedance.com

Chao Peng*
ByteDance
Beijing, China
pengchao.x@bytedance.com

Bo Jiang
ByteDance
Shenzhen, China
jiangbo.jacob@bytedance.com

***Abstract*—Large Language Models (LLMs) have become increasingly integral to enhancing developer productivity, particularly in code generation, comprehension, and repair tasks. However, fine-tuning these models with high-quality, real-world data is challenging due to privacy concerns and the lack of accessible, labeled datasets. In this paper, we present DialogAgent, an automated tool for generating synthetic training data that closely mimics real developer interactions within Integrated Development Environments (IDEs). DialogAgent enables the production of diverse, high-fidelity query-response pairs by simulating multi-turn dialogues and contextual behaviors observed in real-world programming scenarios. The tool significantly reduces the reliance on manual data generation, increasing efficiency by 4.8 times compared to traditional methods. Our experiments and online deployment demonstrate substantial improvements in model performance for code-related question-answering tasks: the acceptance rate of responses generated by our in-house model is improved by 33%, after training on synthesized data generated by DialogAgent.**

***Index Terms*—Agent, Large Language Models, Code Question Answering**

I. Introduction

With the rise of Large Language Models (LLMs), a new generation of LLM-based programming assistant tools has emerged, providing developers with powerful capabilities such as code completion [1]–[4], debugging [5]–[7], and natural language-based code understanding [8]. These tools have become indispensable for enhancing developer productivity, offering support in real-time within Integrated Development Environments (IDEs) and during various stages of software development.

However, deploying these programming assistants in real-world scenarios presents significant challenges. The security and privacy of user data are of paramount importance, particularly when dealing with proprietary code and sensitive information. This makes it impractical to rely on closed-source commercial APIs, which may expose sensitive data to external services. Additionally, the demand for faster inference speeds further constrains the deployment of large LLMs, as models must balance computational efficiency with parameter size to ensure usability in local or private environments. As a result, there is a growing need for personalized smaller-sized LLMs that can be optimized for specific user and business contexts, all while adhering to strict privacy policies.

A key challenge in developing such personalized LLMs lies in the training data [9]. While real-world data is invaluable for fine-tuning LLMs, its use in training is often restricted by user agreements and internal security policies. This makes it impossible to leverage real user data in scenarios involving LLM-based programming assistants. Thus, generating high-quality synthetic training data that closely mimics real-world development environments becomes crucial for training models that can operate effectively in these contexts. The synthetic data must reflect the diversity of real-world interactions while allowing precise control over data categories such as business use cases, user perspectives, and code problem scenarios.

To address these challenges, we have developed DialogAgent, a tool designed to efficiently generate synthetic training data that simulates real IDE environments and developer interactions. Our tool categorizes the data based on critical dimensions such as business scenarios, user intent, reference regions for answers, and the nature of the code-related problems being addressed. This enables the construction of targeted, high-fidelity synthetic datasets that are well-suited for fine-tuning LLMs in a secure and controlled manner.

Additionally, we introduce and compare manual and automated evaluation standards to assess the effectiveness of the generated data. These evaluations are tailored to business-specific outcomes, ensuring that the synthetic data not only meets general quality standards but also aligns with practical development needs. Experimental results demonstrate that DialogAgent is able to generate questions reflecting real-world user queries and high-quality answers compared to human annotators. In addition, after training on the data produced by DialogAgent, the acceptance rate of our internal is increased by 33% which is used by our developers for daily code-related

*Corresponding author.

questions answering.

In this paper, we aim to demonstrate the potential of synthetic data generation in overcoming the limitations of real data usage while maintaining high relevance to real-world development scenarios. Our findings contribute to the ongoing advancement of LLMs in software development, particularly in enhancing their effectiveness within secure, personalized programming assistants.

II. Background and Related Work

A. Large Language Models

Large language models (LLMs) are powerful pre-trained models designed to process and generate natural language. These models undergo two key training phases: first, they are pre-trained on vast amounts of text in an unsupervised manner, learning general linguistic patterns and structures; then, they are fine-tuned for specific tasks to optimize their performance on particular applications.

LLMs used for code-related tasks can generally be grouped into three categories based on their architecture: encoder-only models, decoder-only models, and encoder-decoder models [10]. These models, often built on the transformer architecture, are known for their superior ability to learn from large datasets and scale effectively to handle complex tasks.

- **Encoder-only models**, such as CodeBERT [11], rely on a bidirectional transformer encoder and attention mechanisms to learn vectorized embeddings of input code sequences. These models are particularly suited for tasks that require understanding and representation of code, such as code classification, code clone detection, and code search, where generation is not the primary objective.
- **Decoder-only models**, like CodeGen [12], InCoder [13], and Codex [14], utilize an autoregressive transformer decoder to generate code sequences. These models excel at open-ended tasks such as code generation, where the goal is to produce new code from a given input prompt, making them ideal for scenarios like autocomplete or writing code from natural language descriptions.
- **Encoder-decoder models**, such as LEAM [10], combine both an encoder and a decoder, making them versatile for tasks that involve both understanding and generating code. These models are effective in a range of applications, including code completion, summarization, and generation, allowing them to handle more complex, multi-stage tasks in development workflows.

B. Training Data for Code LLMs

Regardless of their architectural differences, most LLMs can be fine-tuned with task-specific datasets to improve performance on particular tasks. As a result, the need for high-quality training data has become more critical than ever [9], [15], [16]. In this paper, we focus on supervised fine-tuning (SFT) on query-response pairs derived from real-world developer interactions, which is promising to enhance the performance of LLMs in code-related tasks. However, obtaining this data remains a significant challenge.

There have been two main methods of collecting such training data: human annotation and automated data generation [17]–[22]. While human annotation provides high-quality, contextually relevant data, it is a time-consuming and costly process that limits scalability. On the other hand, existing automated approaches often fall short of capturing the diversity and complexity of real developer interactions, resulting in synthetic datasets that do not fully represent real-world behaviors in integrated development environments (IDEs). These limitations hinder the effectiveness of LLMs when applied to practical coding scenarios, where diversity in query types, code contexts, and problem-solving approaches is critical for model success.

C. Motivation

IDE users frequently interact with code completion, explanation, and repair tools embedded within the environment. These interactions often involve multi-step, context-driven queries that span across different programming languages, code files, and even past interactions in the same session. Therefore, capturing this rich context, is paramount for collecting training data, representing diverse real-world scenarios.

III. DialogAgent

In this section, we introduce DialogAgent, a novel method designed to automatically generate SFT data at the repository level with a high degree of diversity, quality, and fidelity to real IDE usage. Figure 1 illustrates the workflow of DialogAgent, which leverages the Q&A plugin within the VS Code IDE and automates the interaction via a UI automation tool.

Our method begins with a set of existing online user queries, a seed chat model (DeepSeek-Coder-33B [23]), and the absence of compliant supervised training data that closely reflects the day-to-day needs of developers. DialogAgent generates synthetic data by simulating multi-agent interactions within real development environments. The LLM-based agent acts in three roles: (1) a Q&A Developer Behavior Analyst (QA-DBA), analyzing and modeling developer behavior in the IDE, and producing a data generation plan; (2) a chat configuration generator, which constructs detailed conversation configurations (e.g., code context, query, file operations); and (3) a response generator, which filters and selects high-quality responses from a pool of candidate outputs.

A. QA Developer Behavior Analyst

The process of QA-DBA is shown in Figure 2. To ensure the generated data is diverse and reflects real online interactions, we first conduct a thorough analysis of developer behaviors using our QA-DBA module. This agent performs a comprehensive review of Q&A interactions within IDEs, capturing 10 key behavioral dimensions across more than 40 categories as shown in Table I, of which the first 7 are classified by the Rule Matcher, and the last 3 are classified by the LLM classifier with the prompt in Figure 3. This analysis informs a data production plan, tuned to reflect online developer behavior patterns, as shown in Figure 4. The QA-DBA framework

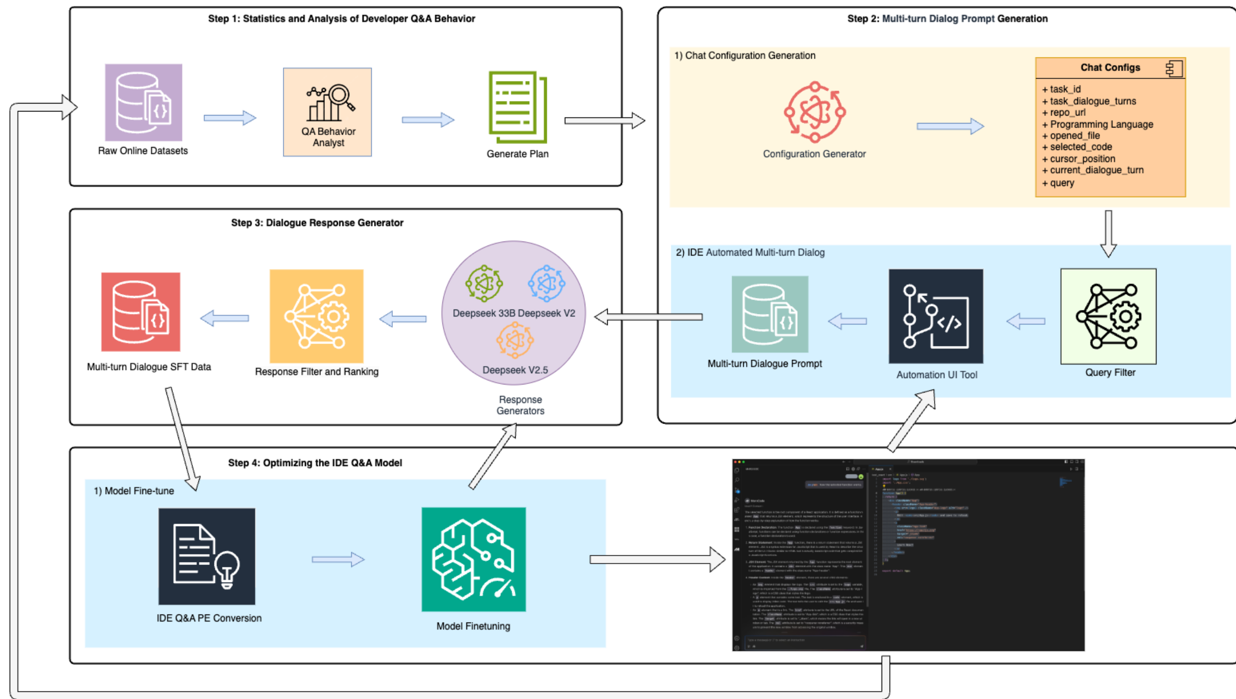


Fig. 1: Workflow of DialogAgent

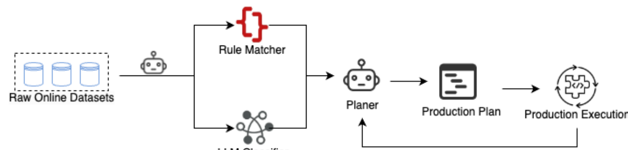


Fig. 2: Process of QA-DBA

LLM-as-a-Behavior-Classifier Prompt

You are a code expert, your task is to make a judgment the intent and difficulty level of the question for the QA pair and to determine the information that is to be the main reference for answering the question.
Standards for determining the intent of a question:
The possible intents of a multi-turn dialogue question are as follow, please directly output the category to which the question belongs, and only one category should be output, for example:
Question: Generate a Fibonacci sequence that sums up to three numbers. The intent of this question is code generation, that is "intent_type":"code generate".
- Code Editing: Add, delete and change the existing code, for example: code optimization, code simplification;
- Code Generation: New generation to meet certain requirements of the code, for example: generation of fast row;
- Code General QA: General R&D knowledge or code knowledge of the problem, for example: the use of vscode;
- Code Explanation: Explanation or introduction to the code , for example: explain the code;
- Comment Generation: Comment of the code, for example: add comments to this code;
- Code Code Repair: Bug fixes to the code, for example: please fix the code;

Standards for question difficulty level:
- Elementary: These questions usually deal with basic programming concepts such as variables, data types, control structures (loops, conditional statements) etc. They usually require only a basic knowledge of the programming language to answer.
- Intermediate: These questions may involve more complex topics such as functions, object-oriented programming, data structures (e.g., lists, dictionaries, trees), exception handling, and basic algorithms. Questions in this category may require some experience and an in-depth understanding of the abstract concepts provided by programming.
- Advanced: These problems may involve more complex topics such as concurrent and multithreaded programming, memory management, operating system-related topics, network programming, design patterns, advanced algorithms, and so on. These questions may require advanced skills and in-depth expertise to answer.
- Expert: These types of questions may involve specialized areas of knowledge or tools, such as specific software architectures, frameworks, libraries, cloud computing, artificial intelligence, machine learning, big data processing, and so on.

Standards for the response reference region:
- Only the names of the following references can be output, the names cannot be created out of thin air or modified: 'Historical Dialog', 'Selected Code', 'Context', 'Question', 'Error Messages', 'General Knowledge';
- If more than one reference is made at the same time, the name of the main referenced section is output;
What's in the reference region for the question:
Historical Dialog:
{history}
Selected Code:
{selected_code}
Context:
{context}
Error Messages:
{error_code}
Question:
{query}

Please return the intent, the difficulty level and the reference region for the question, as well as its reasoning as a json with 6 keys:
"intent_reason"
"intent"
"difficulty_level_reason"
"difficulty_level"
"reference_region_reason"
"reference region"

Json format of intent reasoning, intent, difficulty level reasoning, difficulty level, reference region reasoning, reference region:

Fig. 3: Prompt for Behavior Judgment.

ensures that the generated training data mirrors real-world developer interactions, maintaining consistency and diversity across different query types, code contexts, and development tasks. The LLM classifier and planner are both based on the Deepseek-Coder-V2-Instruct [24] model.

LLM-as-a-Data-Production-Planner Prompt

You are a data production planner, given a situation of online QA developer behavior data statistics, which contains more than 40 sub-categories under 10 latitudes, please output a data production plan that is close to online situation.
Online QA developer behavior statistics.
{statistics}
Number of production data:
{number}
Output a data production plan in json format, with each piece of production data outputting the fine-grained classification under the 10 latitudes to which it belongs:

Fig. 4: Prompt for Data Production Planning.

B. *Chat Configuration Generator*

The chat configuration generator takes as input the interaction context based on developer behavior patterns analyzed by QA-DBA. These configurations define the repository setup (e.g., cursor position, selected code) and trigger mechanisms (e.g., query type, dialog turn) within the IDE (as illustrated in Figure 7). Based on the generated configuration with *Cursor Behavior* and *Programming Language* definitions, an open-source repository from BigCode [25] randomly selected to craft queries. A sample query generation prompt is illustrated in Figure 5, which incorporates various attributes such as *Query Locale*, *Intent*, and *Difficulty Level*. A sample of the generated query is in Table II. In the final phase, we harness the prompt illustrated in Figure 6 and pass it through Deepseek-Coder-V2-Instruct [24] that filters out the queries based on their quality. These configurations ensure that the generated data reflects diverse development scenarios, resulting in high-fidelity training data that closely mirrors real-world developer interactions.

TABLE I: QA Developer Behavioral Categories

Latitudes	**Categories**
Cursor Behavior	no active file, have active file, select a block, select multiple blocks, select a line, select multiple lines
Triggering Q&A Method	inline chat, chat view
Instruction Type	query, quick chat template + query, quick chat template
Programming Language	python, go, cpp, java, javascript, typescript, etc.
System Locale	Chinese, English
Dialog Turns	1, 2, 3, 4, etc.
Locale Requirements in Query	different from system locale, same with system locale, no requirement
Response Reference Regions	historical dialog, selected code, context, question, error messages, general knowledge.
Difficulty Level	elementary, intermediate, advanced, expert
Intent	code generation, code editing, code explanation, comment generation, code repair and code general Q&A

TABLE II: Samples of the Generated Query

Intent	**Difficulty Level**	**Response Reference Regions**	**Query**
Comment Generation	Elementary	Selected Code	Generate comments for the currently selected code.
Comment Generation	Intermediate	Historical Dialog	Any specific comments needed for the 'new' method in Rust to clarify its purpose and usage?
Code Editing	Advanced	Context	Add new code logic, xlsx_file_path is set as a multi-line string, get the path and then loop through xlsx_to_txt for each path.
Code Editing	Expert	Selected Code	Please refactor this code to optimize it for best performance.
Code Explanation	Elementary	Selected Code	Can you explain the significance of 'runApp(MyApp())' in the context of this app's structure?
Code Explanation	Intermediate	Historical Dialog,	Can you go into more detail?
Code Repair	Advanced	Question	init(android.os.Parcel) failed to verify. What's the problem?
Code Repair	Expert	Error Messages	Query: Please fix the code. Error Messages in prompt: Code: use rand::Rng; Error Messages: unresolved import 'rand'; use of undeclared crate or module 'rand'
Code Generation	Elementary	General Knowledge	Write a bubbling sort algorithm.
Code Generation	Intermediate	Historical Dialog	Continue to generate subsequent.
Code General Q&A	Advanced	Question	Using python to solve the equation dy/dx=5y+2x.
Code General Q&A	Expert	General Knowledge	Make a python drawing software, list the libraries you use, give demos.

LLM-as-a-Query-Generator Prompt

```
You are a code expert, please generate a question related to the selected snippet of a piece of code, {%- if reference selected code %}{which is related to {intent}}{% endif %}{%- if reference historical dialog %}{and supplemented with historical dialog to generate the next round of {intent} related question}{% endif %}. The current repository language is {language}, so please ask the question in {locale} and keep it concise and colloquial.
### Instruction requirement
{instr_requirement}
### Question style
Please try to mimic the style of the following questions, using short statements to indicate your intent.
{instr_type}
### The current complete code
{code}
{%- if selected_code %}
### Selected code
{{selected_code}}{% endif %}
{%- if historical_dialog %}
### Historical dialog
{{historical_dialog}}{% endif %}
Please generate a question related to the selected code, taking into account the requirements of the instruction and mimicking the style of questioning.
```

Fig. 5: Prompt used by the query-generating.

LLM-as-a-Query Filter Prompt

```
You are a code expert and your task is to make quality judgments on user questions and filter out poor quality data. Given a code-related multi-turn dialogue question, please evaluate the quality of the user question. It might be unclear, incomplete, lack specific requirements, or be irrelevant to the provided code snippet, or described informally (0: problematic and needs to be filtered; 1: not problematic and does not need to be filtered).

A question that aligns with the following conditions does not need to be filtered, output is 1:
- Development related issues: Project related, code related, development related, software knowledge related.
- Identifiable intention of the question, including: code generation, code editing, code explanation, comment generation, code repair and code general Q&A.

If any of the following quality issues exist in the question, it needs to be filtered, output is 0:
- Non-developmental question. For example: "What can you help me with?"
- Both question and answer provided, questions that have already been answered, or questions that detail the above code.
- Redundant description, repeatedly asked the same question, or identical lines of text.
- Unclear intention in the question. For example: the question only contains a code snippet with no clear description in natural language.
- The question is an extended one based on the historical question but the historical question is empty.

Example: Question: "You", this question is unclear in its intention and unrelated to development issues, so "question_score": 0

Please return scoring and reasoning in JSON format with two keys:
"question_score"
"question_score_reason"

Related code snippet:
{code}

Historical questions:
{question_list}

Question:
{question}

Scoring and reasoning in JSON format
```

Fig. 6: Prompt used by the query quality filter.

C. UI Automation in the IDE Plugin

Due to the complexity of replicating real IDE environments and the lack of a low-cost solution to operate the IDE via APIs, we employ a UI automation tool to simulate user interactions within the IDE. This tool executes predictable tasks such as selecting files, positioning cursors, and triggering the Q&A interface, based on the chat configurations. The Q&A triggering process is shown in Figure 7. According to the input chat configurations, the UI automation tool will first pull down the repository and open the target file, the cursor position will first determine whether the code is selected or not, and then determine whether to move the cursor to the target position. In the next step, the tool will trigger the online Q&A process by clicking on the icon to get to the Q&A interface and entering the query in the dialog box. It captures the resulting Q&A interactions, which are stored and iteratively refined for further use in model training. The use of UI automation not only ensures data diversity but also dramatically reduces the manual effort required for generating large-scale datasets.

Details of the UI automation process are shown in pseudo code in Algorithm 1.

Algorithm 1 Plugin UI automation

```
Input: Chat Configurations
 1: Pull the target code repository;
 2: Open the target file;
 3: task_dialogue_turns ← number of rounds of dialogue;
 4: for t in task_dialogue_turns do
 5:     if selected_code_start_line_index
        and selected_code_end_line_index then
 6:         Selected the target code;
 7:     else if cursor_position_start_line_index
        and cursor_position_end_line_index then
 8:         Move the cursor to the target position;
 9:     end if
10:     Open the marscode Q&A interaction page;
11:     Type in a query to trigger an online Q&A process;
12:     Capture the necessary information from the trace file;
13:     Store the Q&A information in the database;
14: end for
Output: Multi-turn Dialogue Prompts
```

D. Response Generator

After generating diverse and complex queries, high-quality responses are crucial for producing a useful training dataset. To achieve this efficiently, we employ a multi-model response generation pipeline using models from the DeepSeek series [24], [26]. The pipeline comprises two phases: **response generation** and **response judgment**.

1) Response Generation: In the response generation process, we select three LLMs from the same series as our response generators. These include Deepseek-Coder-33B-Instruct [23], Deepseek-Coder-V2-Instruct [24] and Deepseek-V2.5 [26], all of which perform well on code benchmarks such as HumanEval [14] and EvalPlus [27].Each query produced by UI automation is used to prompt all models to generate responses, forming a set of candidate responses for training. In the next phase, scoring will be conducted to select the best response for each query.

However, not every query has a perfect response as sometimes all candidate responses are not of high quality. Instructions without an high-quality response will be filtered out, which may lead to the waste of instructions. Therefore, we also optimize the pool of LLMs to enhance the probability of generating perfect responses. This is achieved through a self-improvement method, that is, fine-tuning the models within the pool using the data synthesized from DialogAgent itself.

2) Response Judgment: In the judgment process, we use the LLM-as-a-Judge methodology [28] to annotate candidate responses with GPT-4o. We combine LLM scoring, ranking and rule-based deduction to select the optimal answer for each query:

- **Response scoring.** We use GPT-4o to score each answer from 1 to 5, comprehensively evaluating the ability to follow instructions and the quality of the answer. GPT-4o is asked to first output the rationale, and then produce the score. The prompt used is given in Figure 8.
- **Rule-based deduction.** For response requirements that can be easily judged by rules but where the model has a chance of making incorrect judgments, we employ a rule-based deduction method for scoring. This score will be subtracted from the score obtained in the Response Scoring phase to determine the final score of the answer. Response that ultimately score 5-point will be used as our final training set. The deduction rules are detailed in Table III.
- **Response comparison.** If an instruction has more than one 5-point response, we will use GPT-4o to rank the responses and select the best one for the training set. The prompt used for comparison is given in Figure 9.

We only select top-scoring answers for training, and queries without a 5-point candidate answer will not be included in the training set to ensure the quality of the training data.

IV. Experiment

We investigate the following research questions (RQ) to study the effectiveness and efficiency of DialogAgent:

- RQ1. What is the impact of efficiently expanding high-quality code question-and-answer data on code Q&A systems?
- RQ2. How can synthetic data maintain consistency with real development environments, particularly in multi-turn dialogue data based on historical conversations?
- RQ3. What is the quality of automatically generated code question-and-answer responses?

We discuss evaluation metrics, results and analysis for each research question in the rest of this section.

A. Experiment Setup

We first de-identified real developer Q&A data to ensure compliance with privacy standards. For evaluation, we balanced the dataset according to user intent and programming languages, selecting a random sample of 300 entries. The evaluation process was carried out by developers proficient in various programming stacks who manually scored the model's responses.

The scoring process is shown in Figure 10. Each model's final score is the average of two rounds of inference, with each round going through three stages: initial scoring, quality inspection, and final score confirmation. A team of 21 annotators handled the scoring process: 13 for initial scoring, 6 for secondary quality inspection, and 2 for final score confirmation. We used a penalty points system (Table IV) for scoring, with a maximum score of 5.

We established two key evaluation metrics:

- **Usability Rate (UR):** The proportion of responses deemed usable ($FinalScore \geq 4$).
- **Perfect Score Rate (PSR):** The proportion of responses judged as perfect ($FinalScore = 5$).

We use Deepseek-coder-33B-Instruct as the seed model and employ high-quality Q&A data produced by DialogAgent for supervised finetuning (SFT) , and each SFT model iterates for

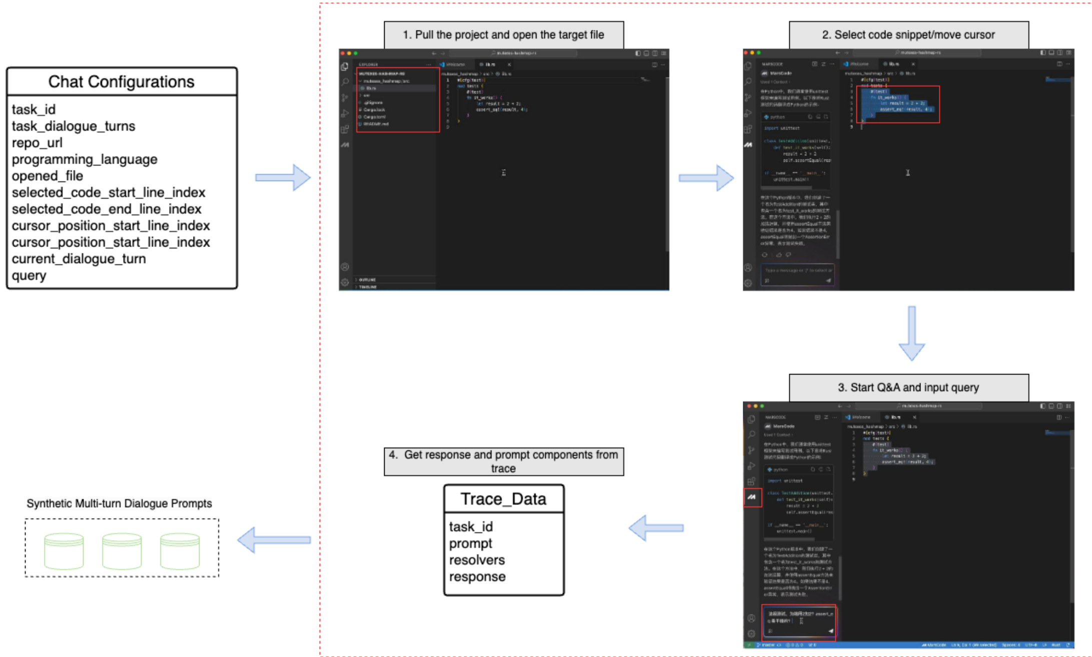
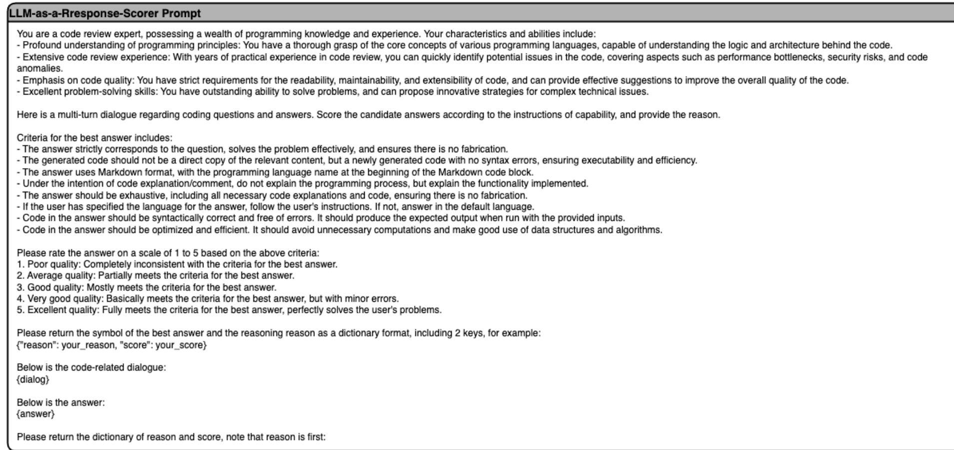


Fig. 7: Framework of the Automation UI Tool



LLM-as-a-Rresponse-Scorer Prompt

You are a code review expert, possessing a wealth of programming knowledge and experience. Your characteristics and abilities include:
- Profound understanding of programming principles: You have a thorough grasp of the core concepts of various programming languages, capable of understanding the logic and architecture behind the code.
- Extensive code review experience: With years of practical experience in code review, you can quickly identify potential issues in the code, covering aspects such as performance bottlenecks, security risks, and code anomalies.
- Emphasis on code quality: You have strict requirements for the readability, maintainability, and extensibility of code, and can provide effective suggestions to improve the overall quality of the code.
- Excellent problem-solving skills: You have outstanding ability to solve problems, and can propose innovative strategies for complex technical issues.

Here is a multi-turn dialogue regarding coding questions and answers. Score the candidate answers according to the instructions of capability, and provide the reason.

Criteria for the best answer includes:
- The answer strictly corresponds to the question, solves the problem effectively, and ensures there is no fabrication.
- The generated code should not be a direct copy of the relevant content, but a newly generated code with no syntax errors, ensuring executability and efficiency.
- The answer uses Markdown format, with the programming language name at the beginning of the Markdown code block.
- Under the intention of code explanation/comment, do not explain the programming process, but explain the functionality implemented.
- The answer should be exhaustive, including all necessary code explanations and code, ensuring there is no fabrication.
- If the user has specified the language for the answer, follow the user's instructions. If not, answer in the default language.
- Code in the answer should be syntactically correct and free of errors. It should produce the expected output when run with the provided inputs.
- Code in the answer should be optimized and efficient. It should avoid unnecessary computations and make good use of data structures and algorithms.

Please rate the answer on a scale of 1 to 5 based on the above criteria:
1. Poor quality: Completely inconsistent with the criteria for the best answer.
2. Average quality: Partially meets the criteria for the best answer.
3. Good quality: Mostly meets the criteria for the best answer.
4. Very good quality: Basically meets the criteria for the best answer, but with minor errors.
5. Excellent quality: Fully meets the criteria for the best answer, perfectly solves the user's problems.

Please return the symbol of the best answer and the reasoning reason as a dictionary format, including 2 keys, for example:
{"reason": your_reason, "score": your_score}

Below is the code-related dialogue:
{dialog}

Below is the answer:
{answer}

Please return the dictionary of reason and score, note that reason is first:

Fig. 8: Response Scoring Template

3 epochs and follows the experimental setup of [23]. During inference, we generated two response variations per prompt using a temperature of 0.3 and top-p of 0.95. The results were averaged over two rounds of inference.

B. RQ1. Impact of High-Quality Code Q&A Data on Code Q&A Systems

In this section, we assess the impact of DialogAgent on six key code Q&A tasks: code generation, code editing, code explanation, comment generation, code repair, and general Q&A. The comparison results between the seed model (DeepSeek-33B-Instuct) and the fine-tuned model (DeepSeek-33B-Instruct-SFT) are shown in Tables V, VI and VII.

We observe significant improvements across all tasks and metrics for the fine-tuned model. For code generation tasks such as code repair, code generation, and code editing, UR increased by 11%, 19%, and 50%, respectively. For code understanding tasks such as code explanation, general Q&A, and comment generation, UR improved by 15%, 2%, and 55%, respectively.

To study the efficiency compared with human annotators, we asked the annotation team to construct code Q&A pairs with the help of online search and other intelligent tools. However, manual construction of code Q&A pairs is time-consuming, yielding only 30 pairs per day per annotator. In contrast, DialogAgent can generate 1,440 pairs per day per instance — 4.8

LLM-as-a-Rresponse-Comparator Prompt

````
You are a coding expert, your task is to evaluate the quality of the candidate answers based on the question, and select the most optimal one among them. You are given a multi-turn dialogue related to coding, from which you are to select the best answer from the candidate answers, and provide a reason for your choice.

Criteria for the perfect answer:
- The answer should strictly align with the question, effectively solving the problem, and ensuring no fabrication or irrelevant content.
- The generated code should not be a direct copy of any existing code, but newly generated, free from any syntax errors, and ensuring executability and efficiency.
- The answer should use Markdown format, ensuring the programming language name is included at the beginning of the Markdown code block, and avoiding wrapping the entire response within ```.
- Under the code explanation/comment intent, do not explain the coding process, but rather the implemented functionality.
- When the code required by the question matches the user's code, the answer should not directly copy the user's code, but generate new code.
- The answer should be comprehensive, including all necessary code explanations and code, ensuring no fabrication or irrelevant content.
- If the user has specified a language for the answer, follow the user's instructions. Otherwise, answer in the default language.
- Code in the answer should be syntactically correct and free of errors. It should produce the expected output when run with the provided inputs.
- Code in the answer should be optimized and efficient. It should avoid unnecessary computations and make good use of data structures and algorithms.

Please return the ranking reason, rank list in which the order of id from good to bad in terms of answer quality, and id of the best answer in JSON format, containing 3 keys:
"reason"
"rank_list:
"best_answer_id"

Note, if the quality of all answers is the same or it is uncertain which answer is the best, fill "best_answer_id" with "NONE".

Below is the code-related dialogue:
{dialog}

Below is the candidate answers:

Answer 1:
{answer_1}

Answer 2:
{answer_2}

Answer 3:
{answer_3}

...

JSON format of the ranking reason, rank list and best answer id:
````

Fig. 9: Response Comparison Template

TABLE III: The Deduction Rules

Scene	Deduction Item	Deduction Score
Inline Chat	Text description before the code	1
Chat View	Lack of basic text description	1
Inline Chat & Chat View	Language of response inconsistent with the instruction request and system setting.	1
	Incomplete code markdown symbols	1
	Altering the original code when editing is no required	2
	Revealing the requirements in the prompt	2
	Incomplete response, truncated in the middle of words or code	5

TABLE IV: The Penalty Points System

Category	Intent	Penalty Points
Content Errors	Common Scene	code errors, description error, response repetition, programming language errors, response truncation, not meeting user requirements
	Special Scene	Code Explanation: nonsense content, missing critical content Comment Generation: nonsense content, missing function header comments, function header comments out of specification, comment locale error Code Repair: missing/redundant code repair Code Editing: missing/redundant code editing
Format Issues	Common Scene	locale error, markdown error, response formatting issues
	Special Scene	Code Repair: lack of complete fix code, lack of basic text description Inline Chat: text description before the code Chat View: lack of basic text description

TABLE V: Manual Scoring Results

Model	PSR	UR
DS-33B-Inst	26.00%	39.00%
DS-33B-Inst-SFT(Ours)	↑46.64%	↑59.40%

times the productivity of human annotators. By scaling computational resources, multiple automation instances can further increase production capacity. Additionally, human inspection also reported that synthetic data generated by DialogAgent demonstrates superior variability and consistency with real-world distributions compared to manually generated data.

C. RQ2. Consistency with the Development Environment

Ww designed 3 groups of ablation experiments to verify the effectiveness of its online consistency, data type increment, and data increment, respectively, and the overall experimental results are compared as in Table VIII, the results of the code generation intention as in Table IX, and the results of the code understanding intention as in Table X.

Synthetic data's consistency with online reality. We generated a batch of data randomly, incorporating elements such as arbitrary programming languages, selected code fragments, and queries, etc. By employing identical training and inference parameters, we fine-tune the Deepseek-coder-33B-Instruct based on the two types of synthetic data, separately. The comparison result is DS-33B-Inst-SFT(Random) VS DS-33B-Inst-SFT(Ours). Evidently, the data generated by DialogAgent overall performs better, an absolute increase of 5% in PSR and 6% in UR. It's noteworthy that in code editing and comment generation, the absolute values of UR have improved by more than 10%.

Data type increment of synthetic data. Multi-turn dialogue capability of a conversational model is important for the performance of IDE code Q&A. However, in the

TABLE VI: Manual Scoring Results in Code Generation Intents

Model	Code Repair		Code Generation		Code Editing	
	PSR	UR	PSR	UR	PSR	UR
DS-33B-Inst	19.23%	44.44%	28.13%	53.13%	22.50%	32.50%
DS-33B-Inst-SFT(Ours)	↑44.44%	↑55.56%	↑56.25%	↑72.00%	↑65.00%	↑82.50%

TABLE VII: Manual Scoring Results in Code Comprehension Intents

Model	Code Explanation		Code General Q&A		Comment Generation	
	PSR	UR	PSR	UR	PSR	UR
DS-33B-Inst	19.23%	38.46%	44.07%	67.78%	16.67%	23.33%
DS-33B-Inst-SFT(Ours)	↑40.46%	↑53.85%	↑50.74%	↑69.26%	↑78.57%	↑78.57%

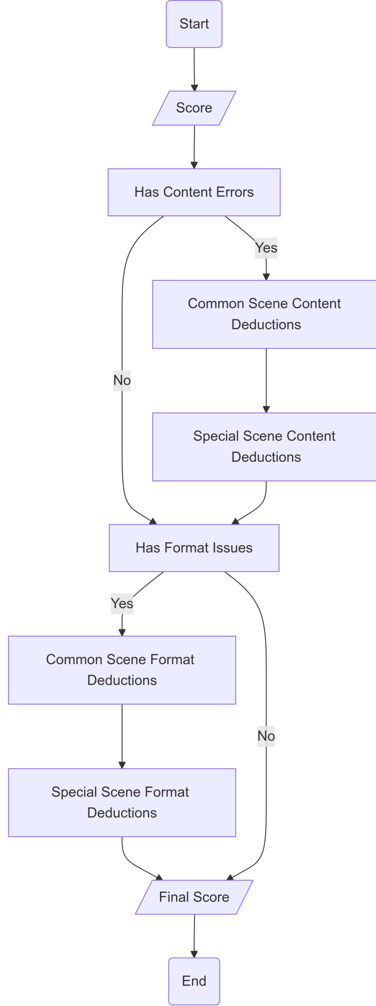


Fig. 10: Process of Scoring

TABLE VIII: Manual Scoring Results for Different Generated Data

Model	PSR	UR
DS-33B-Inst-SFT(Ours)	46.64%	59.40%
DS-33B-Inst-SFT(w/o MTD)	43.10%	52.86%
DS-33B-Inst-SFT(subset)	46.28%	58.11%
DS-33B-Inst-SFT(subset w/o MTD)	42.47%	52.18%
DS-33B-Inst-SFT(Random)	41.47%	53.54%

complex task of generating high-quality multi-turn dialogue data, our DialogAgent provides robust support. We compared the effectiveness of fine-tuned models using synthetic data, considering the presence or absence of multi-turn dialogue data (abbreviated as MTD), i.e., DS-33B-Inst-SFT(Ours) VS DS-33B-Inst-SFT(w/o MTD) and DS-33B-Inst-SFT(subset) VS DS-33B-Inst-SFT(subset w/o MTD). The results indicate that the multi-turn dialogue data generated by DialogAgent can further enhance the performance of the SFT model, with an absolute increase of 3% in the PSR and 6% in the UR. For intents with a stronger historical dialogue dependency (code editing and code generation), there is at least a 7% improvement in UR.

Increment of synthetic data. DS-33B-Inst-SFT(subset) is a subset of the final version dataset DS-33B-Inst-SFT(Ours) with half the data volume. The comparison results indicate that the model with the increased SFT data from DialogAgent can sustain stable effects on the old period evaluation set.

In the same time, the final version of DS-33B-Inst-SFT(Ours) also includes a new batch of data generated based on online data from another time period, and we sampled online data from this time period to serve as a new evaluation data. The comparison results in the new evaluation data are presented in Table XI, XII and XIII. Notably, the model with the increased SFT data generated in the new period performs significantly better on the new period's online data, with PSR achieving a 4% increase, UR a 1% increase.

Given that DialogAgent can autonomously generate data closely resembling online user behavior based on developers' online activity, we can infer that an SFT model based on DialogAgent can achieve continuous automatic optimization.

D. RQ3. Automating High-Quality Code Q&A Responses

To ensure the quality of the generated responses, we evaluate the overall quality and optimization benefits of the response generator, and also assessed the consistency between the automated judgment system and human scoring.

Our annotation team manually score 430 data samples with responses generated by each LLM in our response generator pool. Table XV shows the overall PSR of the LLM pool and the PSR of each LLM in the pool. Although the PSR of each model is not high, when combined together, it reaches 79.76%. Since each model excels in different types of questions, **the approach of using the LLM pool as a response generator**

TABLE IX: Manual Scoring Results for Different Generated Data in Code Generation Intents

Model	Code Repair		Code Generation		Code Editing	
	PSR	UR	PSR	UR	PSR	UR
DS-33B-Inst-SFT(Ours)	44.44%	55.56%	56.25%	72.00%	65.00%	82.50%
DS-33B-Inst-SFT(w/o MTD)	42.78%	58.33%	53.13%	65.63%	61.54%	69.23%
DS-33B-Inst-SFT(subset)	51.11%	61.11%	62.50%	71.88%	55.00%	72.50%
DS-33B-Inst-SFT(subset w/o MTD)	42.78%	52.78%	53.13%	59.38%	55.00%	64.50%
DS-33B-Inst-SFT(Random)	42.78%	52.78%	50.00%	71.88%	60.00%	62.50%

TABLE X: Manual Scoring Results for Different Generated Data in Code Comprehension Intents

Model	Code Explanation		Code General Q&A		Comment Generation	
	PSR	UR	PSR	UR	PSR	UR
DS-33B-Inst-SFT(Ours)	40.46%	53.85%	50.74%	69.26%	78.57%	78.57%
DS-33B-Inst-SFT(w/o MTD)	50.00%	53.85%	59.26%	62.96%	53.33%	63.33%
DS-33B-Inst-SFT(subset)	46.15%	65.38%	59.26%	62.96%	55.17%	68.97%
DS-33B-Inst-SFT(subset w/o MTD)	50.00%	61.54%	48.15%	55.56%	46.67%	63.33%
DS-33B-Inst-SFT(Random)	33.33%	50.00%	57.69%	65.38%	50.00%	66.67%

TABLE XI: Manual Scoring Results in New Evaluation Data

Model	PSR	UR
DS-33B-Inst-SFT(subset)	57.91%	63.92%
DS-33B-Inst-SFT(Ours)	↑61.90%	↑64.17%

combines the strengths of the various LLMs, enabling 79.76% of the data produced in one round has the quality to be included in the training set. Despite this, some prompts still lack perfect answers and will be added to the next production chain until a perfect answer appears. This has to some extent affects the efficiency of our data production. In order to increase the proportion of data that can be used as a training set in each round of data production, we also iteratively optimize the LLM pool through self-improvement method. Table XIV illustrates that after fine-tuning all the models in the LLM pool with the training set from the last production round, the PSR for each intent significantly increases and the average PSR has an increase of 8.65%. The PSR for the Comment Generation intent, particularly, increased by 19.32%.

To evaluate the accuracy of our automated judgment system, our annotation team manually scores 600 responses with distinct instructions. Since the responses we ultimately select for the training set are only 5-point responses, the accuracy rate of scoring 5-point responses is used as a metric to assess our judement system:

$$Accuracy_5 = \frac{pred_5_pos}{pred_5_pos + pred_5_neg} \quad (1)$$

$Accuracy_5$ represents the proportion of 5-point responses scored by our judgement system that align with the scores given by human annotators. It measures the degree of agreement between the judgement system's scoring of 5-point responses and the human evaluations. Also, $pred_5_pos$ indicates instances where both judgement system and human annotation assigned a score of 5, and $pred_5_neg$ represents instances where the judgement system assigned a score of 5, but the human annotation did not.

Considered human scoring as the ground truth, the $Accuracy_5$ for the judgement system is 88.47%. Among the $pred_5_neg$ instances, 96.00% are usable responses, scored more than 3-point by human annotators. 81.62% of the responses manually scored as 5-point are recalled by the system. This indicates that our judgment system has a relatively high consistency with human scoring, and most of the inconsistently scored 5-point responses are still usable.

V. Discussion and Lessons Learned

Industrial deployment confirms the practicality of DialogAgent. DialogAgent is integrated into the model training of our in-house programming assistant, MarsCode, which offers functionalities such as code completion, intelligent Q&A, code generation, explanation, and repair. Along with the launch of DialogAgent's SFT model, the online effectiveness indicator-acceptance rate (percentage of shown answer accepted by the user) was improved by **33%**. The deployment of DialogAgent has demonstrated its practicality and effectiveness in real-world code Q&A scenarios, and also marks the beginning of a continuous improvement cycle. DialogAgent's success in industrial application highlights its role in advancing AI-powered development tools.

Insights from Online User Behavior Through analyzing real-world user interactions, several key behavioral patterns have been identified, which provide valuable insights for further optimizing DialogAgent.

- **Cursor Behavior:** The top 3 cursor behaviors observed among users are: *no active file (40%)*, *having an active file (33%)* and *selecting a code block (35%)*. These behaviors indicate that users often asking questions in a passive browsing state, which highlights the **necessity of automated context retrieval to locate files of interest** to help models better understand the context.
- **Instruction Type:** Although pre-configured quick chat buttons are provided, such as *explain code* and *generate comments*, which are configured with pre-defined prompt templates and will appear when a code snippet is selected by the user, two-thirds of users prefer to directly input their questions instead of using these chat options. This finding highlights the need for natural and flexible query handling in the Q&A tool.

TABLE XII: Manual Scoring Results in Code Generation Intents for New Evaluation Data

Model	Code Repair		Code Generation		Code Editing	
	PSR	UR	PSR	UR	PSR	UR
DS-33B-Inst-SFT(subset)	46.00%	48.00%	80.00%	80.00%	48.00%	60.00%
DS-33B-Inst-SFT(Ours)	↑50.00%	↑50.00%	↑82.00%	↑84.00%	↑56.00%	↑62.00%

TABLE XIII: Manual Scoring Results in Code Comprehension Intents for New Evaluation Data

Model	Code Explanation		Code General Q&A		Comment Generation	
	PSR	UR	PSR	UR	PSR	UR
DS-33B-Inst-SFT(subset)	66.00%	72.00%	54.00%	64.00%	56.00%	64.00%
DS-33B-Inst-SFT(Ours)	↑78.00%	↑78.00%	54.00%	↑68.00%	56.00%	64.00%

TABLE XIV: Manual Scoring Results for Response Generators with Optimization

Model	Code Repair	Code Generation	Code Editing	Code Explanation	Code General Q&A	Comment Generation
	PSR	PSR	PSR	PSR	PSR	PSR
LLM-Pool	72.72%	89.65%	86.48%	94.18%	92.10%	52.27%
LLM-Pool-SFT	↑85.71%	↑96.55%	↑91.89%	↑98.84%	↑94.74%	↑71.59%

TABLE XV: Manual Scoring Results for Response Generators

Model	PSR
LLM Pool (all models)	79.76%
DS-33B-Inst	26.60%
DS-V2.5	64.65%
DS-Coder-V2-Inst	68.14%

- **Dialog Turns:** Most conversations between users and the assistant consist of 5 turns or fewer, with a near $1:1$ ratio of single-turn to multi-turn dialogs. To improve the effectiveness of multi-turn dialogs, it is essential to first enhance the quality of single-turn responses, ensuring that users find value in the tool and continue engaging with it over longer conversations.
- **Response Reference Regions:** When responding to users, the most referenced information sources are the selected code, the user's question, and general programming knowledge. This suggests that focusing on these reference origins can significantly improve the relevance and accuracy of the responses provided by the tool.
- **User Intent Classification:** After classifying the intent of user questions, each type of intent has a dominant subcategory, which accounts for nearly half of the interactions. For example:
  - Code Explanation: Function and method explanations are the most frequent.
  - Code Generation: Test code generation is the most common.
  - General Q&A: Usage of programming language syntax is the most prevalent query.
  - Code Repair: The most triggered is fixing compilation errors.
  - Code Editing: Code translation to another programming language is the most common editing request.
  - Comment Generation: Comments for functions and methods are the most frequent requests.

The Limitation of DialogAgent. Despite the undeniable advantages of DialogAgent, there are certain limitations that need to be addressed. More scenarios expansion requires some development labor. Although it can efficiently handle mainstream code Q&A scenarios, customization and adaptation for more niche or emergent application areas pose a challenge, demanding further manpower and expertise. Additionally, there's a need for a finer classification of scenarios, such as classifications based on knowledge points, e.g., code editing including code optimization, code refactoring, code translation, etc. Such intricate classifications can enrich the diversity of the synthetic data and improve the SFT effect of synthetic data on LLM. Moreover, we will continually broaden and optimize our model pool to keep pace with the rapid development of LLM and meet the ever-evolving requirements of developers. The optimization of the model pool can effectively improve the recall rate of high-quality responses, which in turn greatly improves the robustness and versatility of DialogAgent. This ongoing optimization process underscores our commitment to delivering a tool that stays relevant in the face of dynamic development landscapes and user expectations.

VI. Conclusion and Future Work

In this paper, we explored the challenges and solutions associated with producing high-quality synthetic training data in the scenario of developing LLM-based programming assistants, while ensuring data security and privacy. We introduced DialogAgent capable of efficiently generating synthetic data that mimics real coding scenarios. This tool synthesizes data based on various user perspectives, significantly improving the performance of fine-tuned models in the empirical evaluation. The deployment of DialogAgent to generate training data of our in-house model also experienced a 33% of the improvement of the acceptance rate by our users.

In the future, we consider methods for synthesizing personalized responses and explore more data synthesis schemes and their effects during the reinforcement learning and direct preference optimization phases of model training.

REFERENCES

[1] AnySphere, "Cursor," https://www.cursor.com/, 2024.

[2] "Github copilot," https://github.com/features/copilot, accessed: 2024-05-28.

[3] "Marscode," https://www.marscode.com/, accessed: 2024-05-28.

[4] "Codeium," https://codeium.com/, accessed: 2024-05-28.

[5] Y. Liu, P. Gao, X. Wang, C. Peng, and Z. Zhang, "Marscode agent: Ai-native automated bug fixing," *arXiv preprint arXiv:2409.00899*, 2024.

[6] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1592–1604.

[7] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Agentless: Demystifying llm-based software engineering agents," *arXiv preprint arXiv:2407.01489*, 2024.

[8] S. MacNeil, A. Tran, A. Hellas, J. Kim, S. Sarsa, P. Denny, S. Bernstein, and J. Leinonen, "Experiences from using code explanations generated by large language models in a web software development e-book," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, 2023, pp. 931–937.

[9] G. Dong, H. Yuan, K. Lu, C. Li, M. Xue, D. Liu, W. Wang, Z. Yuan, C. Zhou, and J. Zhou, "How abilities in large language models are affected by supervised fine-tuning data composition," *arXiv preprint arXiv:2310.05492*, 2023.

[10] Z. Tian, J. Chen, Q. Zhu, J. Yang, and L. Zhang, "Learning to construct better mutation faults," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Empowering code generation with oss-instruct," in *Forty-first International Conference on Machine Learning*, 2024.

[16] B. Liu, C. Chen, Z. Gong, C. Liao, H. Wang, Z. Lei, M. Liang, D. Chen, M. Shen, H. Zhou *et al.*, "Mftcoder: Boosting code llms with multitask fine-tuning," in *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2024, pp. 5430–5441.

[17] Z. Tan, A. Beigi, S. Wang, R. Guo, A. Bhattacharjee, B. Jiang, M. Karami, J. Li, L. Cheng, and H. Liu, "Large language models for data annotation: A survey," *arXiv preprint arXiv:2402.13446*, 2024.

[18] J. Park, P. Wisniewski, and V. Singh, "Leveraging large language models (llms) to support collaborative human-ai online risk data annotation," *arXiv preprint arXiv:2404.07926*, 2024.

[19] O. Zendel, J. S. Culpepper, F. Scholer, and P. Thomas, "Enhancing human annotation: Leveraging large language models and efficient batch processing," in *Proceedings of the 2024 Conference on Human Information Interaction and Retrieval*, 2024, pp. 340–345.

[20] S. Bhat and V. Varma, "Large language models as annotators: A preliminary evaluation for annotating low-resource language content," in *Proceedings of the 4th Workshop on Evaluation and Comparison of NLP Systems*, 2023, pp. 100–107.

[21] Y. Yu, G. Rong, H. Shen, H. Zhang, D. Shao, M. Wang, Z. Wei, Y. Xu, and J. Wang, "Fine-tuning large language models to improve accuracy and comprehensibility of automated code review," *ACM Transactions on Software Engineering and Methodology*, 2024.

[22] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in neural information processing systems*, vol. 35, pp. 27 730–27 744, 2022.

[23] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li, F. Luo, Y. Xiong, and W. Liang, "Deepseek-coder: When the large language model meets programming -the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[24] DeepSeek-AI, Q. Zhu, D. Guo, Z. Shao, D. Yang, P. Wang, R. Xu, Y. Wu, Y. Li, H. Gao, S. Ma, W. Zeng *et al.*, "Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence," *arXiv preprint arXiv:2406.11931*, 2024.

[25] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W.-D. Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii, N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. McAuley, H. Hu, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, M. Patwary, N. Tajbakhsh, Y. Jernite, C. M. Ferrandis, L. Zhang, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder 2 and the stack v2: The next generation," 2024.

[26] DeepSeek-AI, A. Liu, B. Feng, B. Wang, B. Wang, B. Liu, C. Zhao, C. Dengr, C. Ruan, D. Dai, D. Guo, D. Yang, D. Chen, D. Ji, E. Li, F. Lin *et al.*, "Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model," *arXiv preprint arXiv:2405.04434*, 2024.

[27] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[28] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. P. Xing, H. Zhang, J. E. Gonzalez, and I. Stoica, "Judging llm-as-a-judge with mt-bench and chatbot arena," *arXiv preprint arXiv:2306.05685*, 2023.