An Agent-based Evaluation Framework for Complex Code Generation

Xinchen Wang[1†], Ruida Hu[1†], Pengfei Gao[2], Chao Peng[2], Cuiyun Gao[1*]
[1]Harbin Institute of Technology, Shenzhen, China
[2]ByteDance, Beijing, China
200111115@stu.hit.edu.cn, 200111107@stu.hit.edu.cn,
gaopengfei.se@bytedance.com, pengchao.x@bytedance.com, gaocuiyun@hit.edu.cn

***Abstract*— Large language models (LLMs) have demonstrated strong capabilities in code generation, underscoring the critical need for rigorous and comprehensive evaluation. Existing evaluation approaches fall into three categories, including human-centered, metric-based, and LLM-based. Considering that human-centered approaches are labour-intensive and metric-based ones overly rely on reference answers, LLM-based approaches are gaining increasing attention due to their stronger contextual understanding capabilities. However, they generally evaluate the generated code based on static prompts, and tend to fail for complex code scenarios which typically involve multiple requirements and require more contextual information. In addition, these approaches lack fine-grained evaluation for complex code, resulting in limited explainability.**

To mitigate the limitations, we propose CodeVisionary, the first agent-based evaluation framework for complex code generation. CodeVisionary consists of two stages: (1) *Requirement-guided multi-dimensional context distillation stage*, which first formulates a detailed evaluation plan by decomposing task requirements, and then stepwise collects multi-dimensional contextual information for each requirement. (2) *Fine-grained scoring and summarization stage*, which defines self-directed and negotiation-based actions, allowing multiple judges to comprehend complex code from fine-grained and diverse viewpoints, and reach a consensus through discussion. A comprehensive evaluation report is also generated for enhanced explainability. For validation, we construct a new benchmark consisting of 363 samples spanning 37 coding scenarios and 23 programming languages. Extensive experiments demonstrate that CodeVisionary achieves the best performance among three baselines for evaluating complex code generation, outperforming the best baseline with average improvements of 0.217, 0.163, and 0.141 in Pearson, Spearman, and Kendall-Tau coefficients, respectively. The resources of CodeVisionary are available at https://github.com/Eshe0922/CodeVisionary.

***Index Terms*—Code generation evaluation, large language models, AI agent**

I. Introduction

With the rapid development of large language models (LLMs), these models have demonstrated promising results in code generation [1], [2]. LLM-based code assistants, such as GitHub Copilot [3] and Cursor [4], have attracted a great number of users, effectively addressing their programming needs. Effectively evaluating the capabilities of LLMs in accomplishing code generation tasks is beneficial for identifying their shortcomings, ultimately enhancing the practical utility in real-world software development scenarios. Current approaches for evaluating code generation can be categorized into human-centered, metric-based, and LLM-based. Human-centered approaches [5], [6] rely on domain experts to evaluate generated code based on their professional knowledge and programming experience. Although domain experts can offer accurate evaluations, these approaches are labour-intensive. Metric-based approaches [7]–[11] typically require reference answers as ground truth or high-quality unit tests, which are generally difficult to create, maintain, and scale across diverse programming languages [12], [13].

Recently, LLM-based approaches [14], [15] have gained increasing attention among researchers and practitioners. LLMs possess strong context-understanding and instruction-following capabilities, allowing them to perform more accurate evaluations. Compared to metric-based approaches, LLM-based approaches do not depend on specific benchmarks, eliminating the need to develop or identify suitable benchmarks, define precise evaluation metrics, or construct ground truth for tasks in the benchmark. However, current LLM-based approaches still face limitations for complex code scenarios. To better demonstrate these limitations, we analyze the failure cases of ICE-SCORE [14], an advanced and representative LLM-based approach for evaluating code generation:

(1) Lack of multi-dimensional contextual information: Complex code scenarios often encompass diverse requirements on runtime behavior, user interaction, and so on. Such requirements often rely on specific contextual information to be properly evaluated. Figure 1(a) illustrates several scenarios where the existing LLM-based approaches fall short. **Scenario ①: Lack of latest programming technology knowledge.** When code generation tasks involve the latest programming technologies, these approaches fail to provide accurate evaluation due to a lack of relevant knowledge. For instance, a task requires using the "`map`" function in "`Python 3.14`" to multiply two lists, with the built-in parameter configured to check for length mismatches. The generated code correctly leverages the "`map`" function with a "`strict`" parameter (Line 6), which is introduced in "`Python 3.14`" to enforce length checking. However, ICE-SCORE assigns a score of

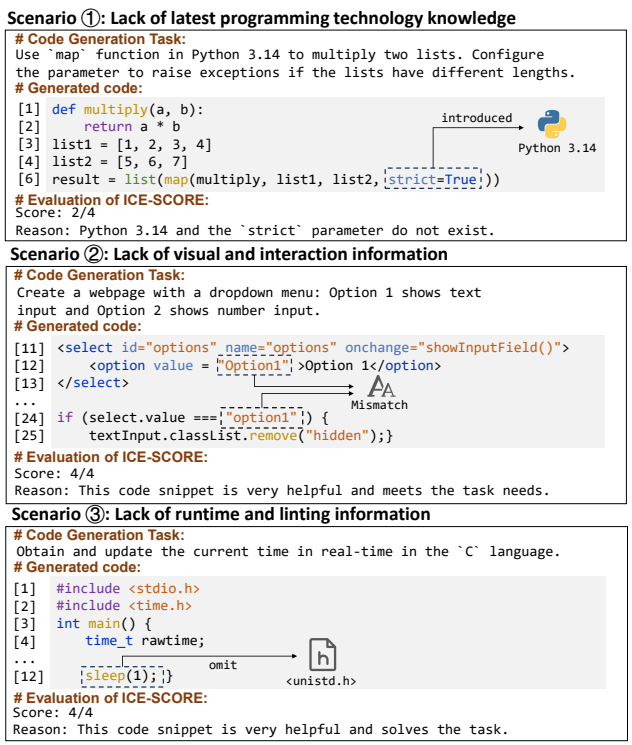
†Work done during an internship at ByteDance.
*Corresponding author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00200

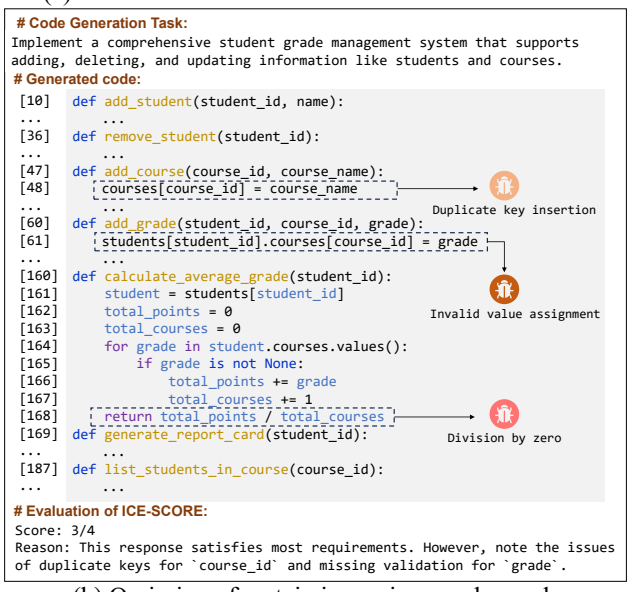
2, reasoning that both "Python 3.14" and the "strict" parameter do not exist. This instance highlights the inability of current LLM-based approaches to handle tasks involving the latest programming technologies. **Scenario ②: Lack of visual and interaction information.** When involving front-end code, these approaches cannot view or interact with the graphical interface, leading to inaccurate evaluations. For instance, a task involves creating a webpage with a dropdown menu. In the generated code, the "select" component's "option" element assigns the value "Option1" (Line 12), while the conditional logic (Line 24) checks for "option1". This mismatch in case between "Option1" and "option1" prevents the intended interaction functionality. However, because ICE-SCORE cannot simulate the graphical interface, it overlooks this issue and incorrectly evaluates the code as meeting the task requirements. This instance highlights that the lack of visual and interaction information may lead to flawed evaluations. **Scenario ③: Lack of runtime and linting information.** Other information regarding code execution, syntax checking, and potential code issues is crucial for comprehensive evaluations. Without external tools, LLM-based approaches cannot access this information [12]. For instance, a task requires generating code in "C" programming language to update the time in real-time. The generated code uses the "sleep" function (Line 12) but omits the necessary "<unistd.h>" header file. This omission would result in a compilation error. As ICE-SCORE cannot execute the code or leverage static analysis tools, it fails to detect this problem and incorrectly assigns a score of 4. The examples indicate that the absence of multi-dimensional context tends to render LLM-based approaches inaccurate [12].

(2) Omission of certain issues in complex code: Real-world code generation tasks often involve multiple requirements, resulting in generated code comprising numerous snippets with potential issues. Current LLM-based approaches struggle to fully comprehend and evaluate complex code in a fine-grained manner, as they typically require multiple reasoning steps and detailed evaluation of each task requirement and its corresponding code snippet [6], [16]. Figure 1(b) illustrates a case of a student grade management system task. The generated code includes numerous functions, such as "add_student", "remove_student", and "calculate_average_grade". ICE-SCORE scores the code 3, deeming it satisfies most task requirements and pinpointing issues such as duplicate keys for "course_id" (Line 48) and missing validation for "grade". However, it overlooks another critical issue in the "calculate_average_grade" function (Line 160). Specifically, the return statement "return total_points / total_courses" (Line 168) does not handle division by zero when "total_courses" is zero. This instance demonstrates that LLM-based approaches may overlook certain issues of complex code.

Besides, current LLM-based approaches solely provide an evaluation score without assessment details, which limits their usefulness in practical development workflows. As the automation of code generation evaluation is being integrated into the Continuous Integration/Continuous Deployment (CI/CD) pipeline of software development [12], real-time and comprehensive evaluation reports are essential for the rapid improve-



(a) Lack of multi-dimensional contextual information



(b) Omission of certain issues in complex code

Fig. 1: Examples for illustrating the limitations of LLM-based approaches for evaluating code generation. Each example includes the code generation task, the generated code, and the evaluation of ICE-SCORE. ICE-SCORE ratings range from 0 to 4, with higher scores indicating higher quality.

ment of LLMs and better align with developers' preferences.

Considering the limitations faced by the LLM-based approaches, we propose an agent-based evaluation framework for complex code generation, named **CodeVisionary**. Agent-based frameworks can interact with the environment by invoking external tools and are capable of handling complex tasks [17]–[19]. Given a code generation task and the corresponding generated code, CodeVisionary evaluates the code through two stages, ultimately providing an evaluation score and evaluation report: **(1) Requirement-guided multi-dimensional context distillation stage**: We first formulate a detailed evaluation plan by decomposing task requirements, and then gather multi-dimensional contextual information for each requirement step by step. **(2) Fine-grained scoring and summarization stage**: We define self-directed and negotiation-based strategies, enabling multiple judges to analyze complex code from fine-grained and diverse perspectives and reach consensus through discussion. Finally, we consolidate the assessment details and construct a structured evaluation report.

We summarize the major contributions of this paper as follows:

(1) We propose the first agent-based evaluation framework for complex code generation, named CodeVisionary, to address the limitations of lacking multi-dimensional contextual information and complete analysis.

(2) We design a two-stage code evaluation pipeline for CodeVisionary: a requirement-guided multi-dimensional context distillation stage for collecting contextual information, and a fine-grained scoring and summarization stage for assessing generated code in a fine-grained manner. Finally, we provide comprehensive evaluation reports.

(3) We conduct thorough experiments on CodeVisionary. The results demonstrate its effectiveness across different programming languages and coding scenarios.

The remaining sections of this paper are organized as follows: Section II presents the architecture of CodeVisionary. Section III describes the experimental setup, including datasets, baselines, and experimental settings. Section IV reports and analyzes the experimental results. Section V further presents the effectiveness of CodeVisionary among different coding scenarios and programming languages, a case study of generated evaluation reports, performance on less complex benchmarks, and the threats to validity. Section VI introduces the background of code generation evaluation and LLM-based agents. Section VII concludes the paper.

II. Approach

In this section, we describe the overall framework of CodeVisionary. As shown in Figure 2, CodeVisionary consists of two stages: (1) Requirement-guided multi-dimensional context distillation stage for gathering contextual information, and (2) Fine-grained scoring and summarization stage for scoring generated code through discussion between multiple judges. CodeVisionary utilizes LLMs as central control agents to conduct multi-turn interactions with the environment. During each interaction, the LLM agent responds with "thought" and "action", respectively, where "thought" represents its reasoning analysis of the environment feedback and "action" represents the command to be executed. As depicted in the "Agent Runtime" part of Figure 2, the actions are executed within the environment, and the corresponding environment feedback is collected as "observation" to the LLM agent in the next interaction.

A. Requirement-guided Multi-dimensional Context Distillation Stage

In this stage, CodeVisionary utilizes an LLM agent to gather multi-dimensional contextual information, including the latest programming technology knowledge, visual and interaction information, runtime and linting information, and so on. The LLM agent follows the paradigm of "environment construction", "requirement comprehension", "plan formulation", and "stepwise analysis". This paradigm mirrors the way humans address complex problems. Before solving complex problems, people typically decompose the requirements and formulate a plan, then proceed to analyze and solve the problem sequentially to ensure all requirements are satisfied, as recent studies [20], [21] conclude that prompting LLMs to formulate plans before addressing problems is effective. Our designed paradigm is as follows:

*a) **Environment Construction**:* This phase aims to construct a complete and executable programming environment, serving as a necessary preparation for subsequent evaluations. Specifically, the agent configures the environment within a Docker container based on the code generation task and the generated code. The initial Docker container is configured with network settings and our custom external tools. Constructing the environment typically involves setting up the programming language runtime and installing the external dependencies. Figure 2 illustrates a task where the agent sets up the "`Python`" interpreter and installs the "`Pandas`" dependency for data processing. For more complex tasks, additional setup of configuration files and environment variables is required.

*b) **Requirement Comprehension**:* This phase aims to enhance the agent's comprehension of the task requirements and prepare for formulating a thorough evaluation plan tailored to each requirement. Specifically, the agent comprehends the code generation task and decomposes the task requirements. As illustrated in Figure 2(a), the agent breaks down the code task into several detailed requirements: "Read and print file contents", "Handle file reading exceptions", and "Count and save occurrences of words". This decomposition enables the agent to better grasp complex tasks by dividing them into smaller, more manageable components.

*c) **Plan Formulation**:* This phase divides the evaluation of generated code into several steps, which facilitates the verification of each task requirement and reduces the reasoning burden on the agent. Specifically, the agent formulates a detailed evaluation plan. Each step in the plan includes a brief **goal** (*i.e.*, "Goal") and specific **guidance** (*i.e.*, "Guidance") on how to achieve the goal. As illustrated in Figure 2(a), the plan

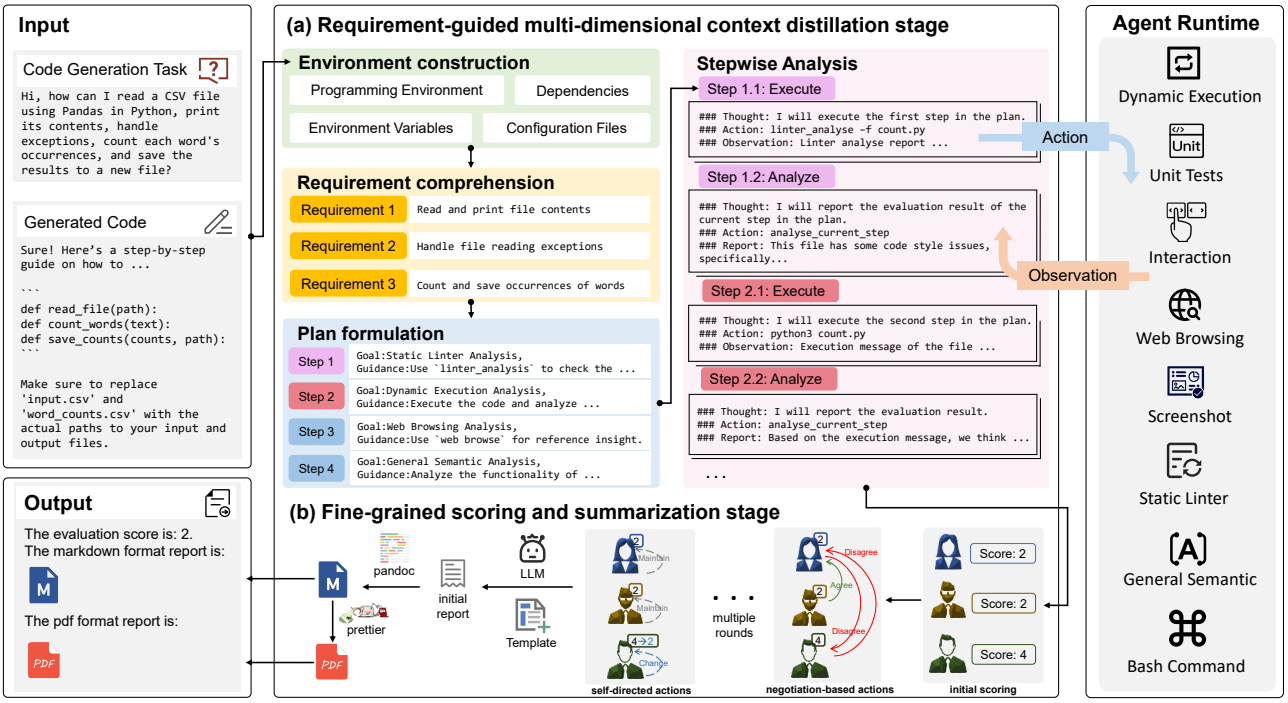


Fig. 2: The architecture of CodeVisionary. It consists of two main stages: (a) Requirement-guided multi-dimensional context distillation stage for collecting contextual information based on the stepwise evaluation plan, and (b) Fine-grained scoring and summarization stage for generating evaluation scores and reports through negotiation with diverse viewpoints.

formulated by the agent includes goals such as "Static Linter Analysis", and "Dynamic Execution Analysis", along with corresponding guidance. The goal and guidance of each step in the plan collectively specify the action that the agent should take. For instance, the goal of the "Static Linter Analysis" is to ensure the generated code adheres to code syntax and style, and the guidance is to take the "`linter_analysis`" action. Below is a list of all possible actions during each step:

- **Dynamic Execution:** This action checks for any compilation errors and analyzes whether the execution message aligns with the expected requirement. Possible commands include "`python test.py`" for running a Python script, "`gcc -o output test.c && ./output`" for compiling and executing a C program, and so on.
- **Static Linter:** This action checks for syntax errors, code issues, and style inconsistencies. It helps identify potential problems in the generated code without execution. We design the command "`linter_analysis -f 'path'`" to check a file or directory for syntax compliance. This command automatically selects appropriate static analysis tools for the given file path and returns a structured analysis report.
- **Unit Tests:** This action involves writing and executing unit tests, ensuring the generated code meets specific requirements and behaves as expected under various conditions. These unit tests support requirement analysis by providing different inputs and validating the outputs through assertions. For instance, consider a function that processes a list of integers and returns a list of their squares, but only for even numbers. A unit test might provide the input "`[1, 2, 3, 4]`" and utilize the assertion "`assert process_list([1, 2, 3, 4]) == [4, 16]`" to verify that the output is as expected.
- **Screenshot:** This action involves capturing a screenshot of the front-end code to support requirement analysis through visualization. We design the command "`screenshot_analysis -f 'path' -q 'query'`", which renders the specified file as a screenshot and generates an analysis report utilizing a multimodal LLM. The agent can also query the screenshot to extract relevant visual information, such as UI elements and layout.
- **Interaction:** This action involves simulating user interactions with the front-end code to evaluate the interaction requirements. We extend the "`screenshot_analysis`" command with the "`-a 'actions'`" parameter to interact with the graphical interface before capturing the screenshot. Possible interactions include clicking elements, filling input fields, hovering over elements, and scrolling.
- **Web Browsing:** This action involves retrieving additional information from the website, including knowledge of the latest and specialized programming technologies. We design the command "`web_browse -q 'query'`" to search the website according to the query.
- **General Semantic:** Considering the LLM's powerful

contextual understanding capabilities, the agent can also evaluate whether the generated code satisfies requirements related to complexity, security, and robustness, without invoking external tools.

- **Bash Command:** In addition to the actions designed specifically for code generation, the agent also needs to call bash commands during the evaluation process to perform steps such as writing code, writing data, viewing files, and navigating the file system.

*d) **Stepwise Analysis**:* In this phase, the agent follows the evaluation plan by executing the specified action at each step and analyzing the corresponding execution results. The agent alternates between executing and analyzing during interactions, with its current state indicated as "`Execute State`" or "`Analyze State`".

- **`Execute State:`** In this state, the agent takes the action specified in the current step. The action is executed within the constructed Docker container, and the corresponding execution results are collected as the environment feedback (*i.e.*, "observation") to the agent. Besides, the following hint is given as additional environment feedback to facilitate the evaluation process:

> **Hint:** *Once you have executed the actions in the current step, you should analyze and report the execution results in the next interaction. If the step is not yet finished, please continue working on it.*

- **`Analyze State:`** In this state, the agent takes the action "`analyze_current_step`" to analyze and report the execution results. We predefine different analysis report templates tailored to the execution results of different actions. Similarly, the following hint is provided as additional environment feedback:

> **Hint:** *You have submitted the analysis report of the current step. Please review the steps you have already completed and proceed to the next step.*

Besides, the agent can adjust the evaluation plan in real time based on the actual evaluation process to enhance flexibility. Finally, the analysis reports from each step, together with the code generation task, generated code, and decomposed requirements, are collected as input for the next stage.

B. Fine-grained Scoring And Summarization Stage

In this stage, CodeVisionary employs multiple LLM agents to act as judges, scoring the generated code through collaborative discussion. Since a single judge may potentially overlook issues in complex code or make misjudgements, involving multiple judges enables more fine-grained evaluation by incorporating diverse viewpoints and facilitating consensus. This negotiation strategy can improve the thoroughness and reliability of the evaluation results, as different judges focus on different issues and aspects of the generated code.

1) Definition of Judges: Let $A_1, A_2, \ldots, A_n$ denote the n LLM agents acting as individual judges. Each judge A_i assigns a score to the generated code based on the decomposed requirements, stepwise analysis reports from Section II-A, and the evaluation criteria. The evaluation criteria include aspects of correctness, functionality, and clarity. Below, we present an example of the clarity aspect and its corresponding evaluation criteria, while the remaining aspects and criteria are available in our repository:

> **Clarity:** *Is the generated code explained in an easy-to-understand manner, logically clear, and unambiguous?*
> **- 0/4:** *Mostly unclear or confusing; very difficult to understand.*
> **- 1/4:** *Significant ambiguity, partially understandable; contains several confusing parts.*
> **- 2/4:** *Some ambiguity, but mostly understandable; contains some unclear parts.*
> **- 3/4:** *Mostly clear, slight ambiguity; generally easy to understand with minor issues.*
> **- 4/4:** *Very clear, no ambiguity, well-organized; completely easy to understand.*

Each judge provides an evaluation score S_i and corresponding scoring reason R_i. Formally, each judge A_i is represented as a tuple:

$$A_i = (S_i, R_i) \tag{1}$$

where S_i and R_i denote the score and reason provided by the judge A_i, respectively.

2) Negotiation and Scoring: After each judge A_i has provided their initial score S_i^0 and reason R_i^0, these evaluations are shared among all other judges, ensuring that each judge is aware of their peers' assessments. This process can be formally expressed as:

$$\forall i, j \in \{1, 2, \ldots, n\}, \quad (S_i^0, R_i^0) \rightarrow A_j \tag{2}$$

Then these judges engage in multiple rounds of negotiation to reach a consensus. Let k denote the round number. In each round k, each judge A_i may take one or more actions $Action_i^k$, which can be categorized into two types: self-directed actions and negotiation-based actions.

Self-directed actions allow a judge to manage their own evaluation, including maintaining the current score S_i and the reason R_i^0 (`Maintain`), changing the score S_i and the reason R_i^0 (`Change`), or withdrawing a previously stated opinion from round k in light of possible opinion changes (`Withdraw`). Formally, these actions are defined as:

$$\text{Actions}_{\text{self-directed}} = \begin{Bmatrix} \text{Maintain}(E) \\ \text{Change}(S_i, R_i, E) \\ \text{Withdraw}(k, E) \end{Bmatrix} \tag{3}$$

Negotiation-based actions, on the other hand, facilitate consensus-building among judges. These actions correspond to three forms of negotiation, including explicitly agreeing with another judge's score and reason (`Agree`), requesting clarification (`Query`), or expressing disagreement (`Disagree`).

TABLE I: Statistics of the constructed benchmark.

Statistics	Number
Samples	363
Code Generation Tasks	121
Coding Scenarios	37
Programming Languages	23
Average String Length	
Code Generation Task	1,906
LLM-generated Response	2,650

Formally, these actions are defined as:

$$\text{Actions}_{\text{negotiation-based}} = \begin{Bmatrix} \text{Agree}(A_j, E) \\ \text{Query}(A_j, E) \\ \text{Disagree}(A_j, E) \end{Bmatrix} \quad (4)$$

Hence, the $Action_i^k$ taken by the judge A_i in round k is given by:

$$\text{Action}_i^k \subseteq \{\text{Actions}_{\text{self-directed}} \cup \text{Actions}_{\text{negotiation-based}}\} \quad (5)$$

These judges are required to provide an explanation (*i.e.*, E) while taking different actions. After each round k, the current score and reason of each judge, as well as the actions taken, are shared among all judges:

$$\forall i, j \in \{1, 2, \ldots, n\}, \quad (S_i^k, R_i^k, Action_i^k) \rightarrow A_j \quad (6)$$

The negotiation ends when either the number of rounds is exceeded or all judges reach a consensus on the evaluation score. The final evaluation score is calculated as the average of the scores provided by all judges:

$$\text{Evaluation Score} = \frac{1}{n} \sum_{i=1}^{n} S_i \quad (7)$$

3) Summarization and Evaluation Report Generation: We obtain the environment configuration, task requirements, and stepwise analysis reports in Section II-A and the final evaluation score in Section II-B2. Besides, CodeVisionary leverages another LLM to conclude the overall evaluation results, which consist of the evaluation reasons and the optimization suggestions provided by multiple judges after negotiation. Formally, this process can be represented as:

$$\text{Evaluation Score}, \{(S_i, R_i) \mid i = 1, 2, \ldots, n\} \longrightarrow \text{Evaluation Reason}, \text{Optimization Suggestion} \quad (8)$$

Then the agent generates the initial evaluation report in Markdown format. To enhance readability, we standardize and beautify the initial evaluation report. We utilize Prettier [22] to check for formatting issues and maintain a consistent style. Besides, we employ Pandoc [23] to convert the Markdown report to PDF format.

III. Experimental Setup

In this section, we evaluate the CodeVisionary and aim to answer the following research questions (RQs):

RQ1: How does CodeVisionary perform in evaluating code generation compared with different methods?

RQ2: What is the influence of different stages on the performance of CodeVisionary?

RQ3: How do different hyper-parameters impact the performance of CodeVisionary?

A. Datasets

CodeArena [24] is a human-curated benchmark of 397 high-quality code generation tasks, covering various coding scenarios and programming languages. It simulates the complexity and diversity of real-world tasks, with each task classified as "easy", "medium", or "hard" based on complexity. To evaluate CodeVisionary's performance across diverse and complex tasks, we filter CodeArena, generate responses using different LLMs, and manually score the responses. The filtering, response generation, and manual scoring process is as follows:

(1) Filtering: We first retain tasks classified as "hard". Then, we filter out tasks that depend on specific software or platforms, such as those requiring MATLAB or Verilog.

(2) Response Generation: We generate responses for each code generation task utilizing three popular LLMs, including GPT-3.5-turbo [25], Claude-3.5-Sonnet [26], and GPT-4o [27].

(3) Manual Scoring: Each code generation task and corresponding LLM-generated response is scored by two experienced experts, each with over five years of expertise in the relevant programming languages. Scoring is conducted on a scale from 0 to 4, where higher scores indicate better response quality, following the widely-used evaluation criteria [5]. Due to page limitations, the detailed evaluation criteria are available in our repository. The Kappa coefficient between the two experts exceeds 80%, indicating a high level of agreement. In case of scoring discrepancies, a third expert is introduced to adjudicate. All experts remain unaware of the identity of the LLM that generates each response.

The statistics of our constructed benchmark are illustrated in Table I. The benchmark covers 37 coding scenarios and 23 programming languages, enabling thorough assessment of CodeVisionary across diverse tasks. Each sample is represented as a triplet: (code generation task, LLM-generated response, human-annotated score).

B. Comparison Baselines

To evaluate the performance of CodeVisionary, we compare the state-of-the-art LLM-based approaches. Following the previous work [15], we also introduce a vanilla approach as the baseline method.

- **VANILLA** directly prompts LLMs to determine the correctness and helpfulness of the response.
- **ICE-Score [14]** utilizes assessment criteria and an evaluation step template to evaluate the usefulness and functional correctness of the response.
- **CODEJUDGE [15]** guides LLMs in performing "slow thinking" to arrive at an in-depth and reliable evaluation of semantic correctness.

TABLE II: Comparison results between CodeVisionary and baseline methods.

Models	GPT-3.5-turbo			Claude-3.5-Sonnet			GPT-4o			**AVG**		
Metrics	r_p	r_s	τ	r_p	r_s	τ	r_p	r_s	τ	r_p	r_s	τ
VANILLA	0.083	0.129	0.117	0.033	0.098	0.094	0.010	0.047	0.045	0.042	0.091	0.085
ICE-SCORE	0.100	0.177	0.157	0.029	0.096	0.092	-0.011	0.054	0.051	0.039	0.109	0.100
CODEJUDGE	0.130	0.130	0.120	0.097	0.074	0.072	0.025	0.078	0.076	0.084	0.094	0.089
CodeVisionary	**0.325**	**0.286**	**0.247**	**0.317**	**0.278**	**0.262**	**0.262**	**0.253**	**0.214**	**0.301**	**0.272**	**0.241**

C. Evaluation Metrics

We follow best practices in natural language generation evaluation and utilize Pearson (r_p), Spearman (r_s), and Kendall-Tau (τ) coefficients to measure the correlation between evaluation scores assigned by different approaches and the ground truth (*i.e.*, scores annotated by humans).

D. Implementation Details

CodeVisionary and the baseline methods are provided access to GPT-4o. For the requirement-guided multi-dimensional context distillation stage, we limit the maximum number of interactions to 40 and set the sampling temperature to 0.2. For the fine-grained scoring and summarization stage, we set the sampling temperature to 0.7 and the number of judges to 3, and allow up to 4 rounds of negotiation. For the baseline methods, we try our best to reproduce them from publicly available source code and papers, and use the same hyper-parameter settings. For all correlation metrics, we use the implementation from SciPy [28] and call these APIs with the default settings.

IV. Experimental Results

A. RQ1: Effectiveness of CodeVisionary in Evaluating Code Generation

To answer RQ1, we conduct a comprehensive analysis against three baseline methods. The experimental results are shown in Table II.

CodeVisionary exhibits superior performance compared with the baseline methods. As shown in Table II, we observe that CodeVisionary consistently outperforms all the baseline methods across three LLMs (*i.e.*, responses generated by GPT-3.5-turbo, Claude-3.5-Sonnet, and GPT-4o). Overall, CodeVisionary achieves average scores of 0.301, 0.272, and 0.241 on r_p, r_s, and τ, respectively, outperforming the best baseline methods with improvements of 0.217, 0.163, and 0.141. These improvements are due to CodeVisionary's ability to collect multi-dimensional contextual information and provide the evaluation score through fine-grained negotiation.

Current advanced LLM-based approaches show minimal difference compared to VANILLA. We observe that state-of-the-art LLM-based approaches, ICE-SCORE and CODEJUDGE, exhibit little difference compared to VANILLA. Specifically, the average scores of VANILLA on r_p, r_s, and τ are 0.042, 0.091, and 0.085, respectively, while ICE-SCORE achieves 0.039, 0.109, and 0.100, and CODEJUDGE achieves 0.084, 0.094, and 0.089, respectively. These results indicate that the differences are marginal. Besides, VANILLA outperforms the other two methods in r_s and τ when evaluating responses generated by Claude-3.5-Sonnet. These results further demonstrate that current LLM-based methods struggle to comprehend complex code without contextual information, which is essential for accurate and comprehensive code evaluation.

> **Answer to RQ1:** CodeVisionary achieves the best performance on evaluating code generation, exceeding the best baseline method by 0.217, 0.163, and 0.141 on r_p, r_s, and τ, respectively.

B. RQ2: Effectiveness of Different Stages in CodeVisionary

To answer RQ2, we explore the effectiveness of different stages on the performance of CodeVisionary.

1) Requirement-guided Multi-dimensional Context Distillation Stage: To understand the impact of this stage, we deploy a variant of CodeVisionary without the requirement-guided multi-dimensional context distillation stage (*i.e.*, w/o RMCD). The variant follows a completely free evaluation process instead of following our designed paradigm of "environment construction", "requirement comprehension", "plan formulation", and "stepwise analysis". As shown in Table III, the addition of the RMCD stage achieves higher performance on r_p, r_s, and τ across all three LLMs. Specifically, adding the RMCD stage improves performance by an average of 27.0%, 32.0%, and 32.4% across the three metrics, respectively. The RMCD stage excels in understanding diverse requirements and formulating detailed evaluation plans, thereby ensuring the thorough collection of multi-dimensional context.

2) Fine-grained Scoring And Summarization Stage: To explore the contribution of this stage, we also construct a variant of CodeVisionary without the fine-grained scoring and summarization stage (*i.e.*, w/o FSAS). This variant scores the LLM-generated response multiple times and takes the average instead of a discussion between multiple judges. As shown in Table III, the addition of the FSAS stage achieves higher performance on r_p, r_s, and τ across all three LLMs. Specifically, adding the FSAS stage improves performance by 21.4%, 21.4%, and 19.9% on average across the three metrics, respectively. The FSAS stage leverages multiple judges for negotiation, mitigating the inaccuracy of comprehending complex code and integrating diverse opinions.

TABLE III: Impact of different stages on the performance of CodeVisionary.

Models	GPT-3.5-turbo			Claude-3.5-Sonnet			GPT-4o			**AVG**		
Metrics	r_p	r_s	τ	r_p	r_s	τ	r_p	r_s	τ	r_p	r_s	τ
w/o RMCD	0.278	0.228	0.200	0.250	0.219	0.198	0.182	0.171	0.147	0.237	0.206	0.182
↪ w/o RT	0.229	0.208	0.185	0.233	0.219	0.193	0.152	0.153	0.133	0.205	0.193	0.170
↪ w/o UI/UX	0.305	0.258	0.217	0.302	0.242	0.216	0.198	0.214	0.169	0.268	0.238	0.201
↪ w/o ST	0.290	0.286	0.234	0.280	0.179	0.157	0.176	0.161	0.136	0.249	0.209	0.176
w/o FSAS	0.282	0.233	0.204	0.262	0.250	0.227	0.201	0.189	0.172	0.248	0.224	0.201
CodeVisionary	**0.325**	**0.286**	**0.247**	**0.317**	**0.278**	**0.262**	**0.262**	**0.253**	**0.214**	**0.301**	**0.272**	**0.241**

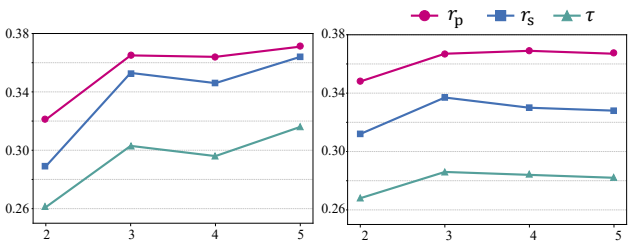
3) Different Information in Requirement-guided Multi-dimensional Context Distillation Stage: To further investigate the influence of different information in the requirement-guided multi-dimensional context distillation stage, we design three variants by disabling different information sources: test execution (runtime information, *i.e.*, RT), screenshot and interaction (UI/UX information), and web browsing and syntax linters (static information, *i.e.*, ST). As shown in Table III, different contextual information contributes to improvements in evaluation accuracy. Incorporating runtime information yields the most notable gains, improving r_p, r_s, and τ by 46.8%, 40.9%, and 41.8%, respectively, highlighting the strong impact of runtime information in capturing execution anomalies and boundary conditions. UI/UX information provides a multi-modal understanding, leading to moderate improvements of 12.3%, 14.3%, and 19.9%, respectively. Other static information from web browsing and syntax linters helps identify potential code issues and provides external context, resulting in gains of 20.9%, 30.1%, and 36.9%, respectively. Overall, these results demonstrate that different types of information each play a complementary role in enhancing evaluation performance.

Answer to RQ2: Both RMCD and FSAS stages can improve the performance of CodeVisionary. The RMCD stage boosts r_p, r_s, and τ by 27.0%, 32.0%, and 32.4%, respectively, while the FSAS stage enhances CodeVisionary by 21.4%, 21.4%, and 19.9%, respectively. Besides, different information in the RMCD is essential to accurate evaluation.

C. RQ3: Influence of Hyper-parameters on the Performance of CodeVisionary

To answer RQ3, we explore the impact of different hyper-parameters, including the number of judges and the maximum number of rounds during the fine-grained scoring and summarization stage.

1) Number of Judges: Figure 3(a) shows the performance of CodeVisionary with different numbers of judges. As the number of judges increases from 2 to 3, the performance improves notably, with r_p, r_s, and τ increasing from 0.321, 0.289, and 0.261 to 0.365, 0.353, and 0.303, respectively.



(a) Number of Judges (b) Maximum Number of Rounds

Fig. 3: The influence of the number of judges and maximum number of rounds on CodeVisionary. The horizontal axis represents the number of judges or the maximum number of rounds.

When the number increases to 5, the performance stabilizes, indicating that further increases have a limited impact. This suggests that three judges are sufficient to provide a comprehensive evaluation, and additional judges struggle to offer new insights. Considering the balance between resource consumption and performance, we choose 3 as the optimal number of judges.

2) Maximum Number of Rounds: Similarly, as shown in Figure 3(b), increasing the maximum number of rounds from 2 to 3 leads to steady performance improvements, with r_p, r_s, and τ rising from 0.348, 0.312, and 0.268 to 0.367, 0.337, and 0.286, respectively. This demonstrates that negotiation allows judges to effectively share insights and mitigate biases in understanding complex code, thereby refining their evaluations. However, as the number of rounds increases from 3 to 5, the performance tends to fluctuate slightly, suggesting that consensus has already been reached and additional rounds bring little to no benefit. To balance resource consumption and performance, we select 4 as the optimal number of rounds.

Answer to RQ3: The performance of CodeVisionary is influenced by the number of judges and the maximum number of rounds. Our default settings of 3 judges and 4 rounds yield optimal results.

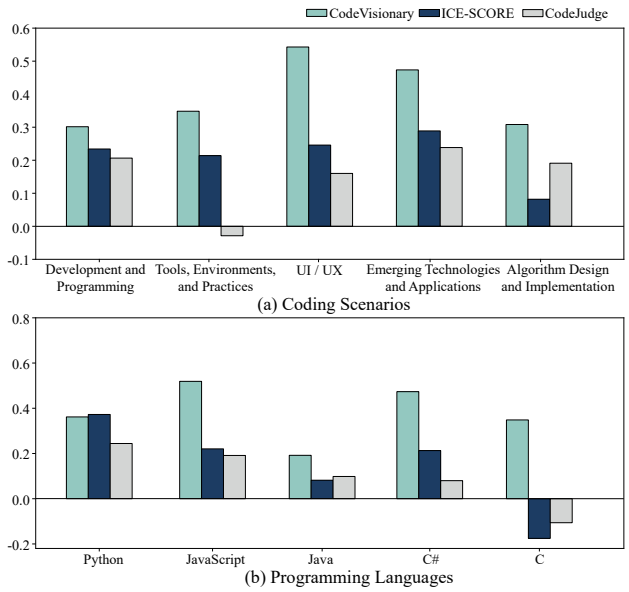


Fig. 4: Performance of CodeVisionary and baseline methods across different coding scenarios and programming languages, measured by r_s.

TABLE IV: Counts and percentages of different actions invoked by CodeVisionary across all instances.

Action	# Total	# Per Instance	%
Dynamic Execution	210	0.58	9.45
Static Linter	366	1.01	16.46
Unit Tests	84	0.23	3.78
Screenshot	76	0.21	3.42
Interaction	53	0.15	2.38
Web Browsing	325	0.90	14.62
General Semantic	652	1.80	29.33
Bash Command	343	0.94	15.43
Other	114	0.31	5.13
All	2,223	6.12	100.00

V. Discussion

A. Performance on Different Coding Scenarios and Programming Languages

Figure 4 illustrates the performance of CodeVisionary and baseline methods across mainstream coding scenarios and programming languages. As shown in Figure 4(a), CodeVisionary generally surpasses baseline methods in different coding scenarios. In particular, CodeVisionary improves the r_s in "UI/UX", "Emerging Technologies and Applications", and "Algorithm Design and Implementation" from 0.246 to 0.543, 0.289 to 0.474, and 0.191 to 0.308, respectively. Table IV presents the counts and percentages of various actions invoked by CodeVisionary. In the "UI/UX" scenario, CodeVisionary employs "Screenshot" (3.42%) and "Interaction" (2.38%) to perform multimodal assessment of front-end code. For the "Emerging Technologies and Applications" scenario, it can leverage "Web Browsing" (14.62%) to acquire the latest or specific programming knowledge. In the "Algorithm Design and Implementation" scenario, CodeVisionary can utilize "Dynamic Execution" (9.45%) and "Unit Tests" (3.78%) to test boundary cases through code execution.

Similarly, Figure 4(b) shows that CodeVisionary generally outperforms baseline methods across different programming languages. It achieves much higher r_s in "JavaScript", "C#", and "C", increasing from 0.220 to 0.519, 0.213 to 0.473, and -0.106 to 0.348, respectively. The strong performance in "JavaScript" can be attributed to the domain information gained from "Screenshot" and "Interaction". Regarding "C", the negative coefficient of baseline methods highlights the importance of execution information. Overall, these results demonstrate that CodeVisionary effectively adapts across coding scenarios and programming languages.

Fig. 5: Example evaluation report generated by CodeVisionary.

B. Case Study of Evaluation Reports

As shown in Figures 5, the evaluation report begins with an overview of the "Code Task", the "LLM-generated Response", and the corresponding "Evaluation Score". The "Environment Configuration" section outlines the environment setup, facilitating reproducible and isolated evaluation. The "Task Requirements" section specifies both overall and specific requirements, enabling targeted assessment of functionality and completeness. The "Stepwise Evaluation Results" section provides a detailed, multi-stage assessment of the generated code. In this example, CodeVisionary utilizes "Dynamic Execution" and "Unit Tests" to identify discrepancies between the code execution output and the task requirements, specifically revealing that the length of the generated list does not match

TABLE V: Comparison of performance on CoNaLa.

Metrics	r_p	r_s	τ
ICE-SCORE	**0.655**	0.596	0.534
CODEJUDGE	0.544	0.491	0.444
CodeVisionary	0.644	**0.637**	**0.572**

the expected length, thereby uncovering functional errors in the code. Besides, CodeVisionary utilizes "Static Linter" to check the code style for potential issues. The "Overall Evaluation Results" section summarizes that the code fails to meet requirements due to major logical errors and offers clear suggestions, such as enforcing the six-selection limit and using more descriptive variable names.

C. Performance on less complex benchmarks

To evaluate the performance of CodeVisionary on less complex tasks, we conduct experiments on the popular CoNaLa dataset [29]. CoNaLa is a Python code generation benchmark consisting of 472 tasks collected from StackOverflow and 2,360 code snippets. We adopt the human annotation collected by Evtikhiev [5] as ground truth. As shown in Table V, CodeVisionary also achieves great performance on CoNaLa. Specifically, CodeVisionary surpasses the best baseline method by 0.041 and 0.038 on r_s and τ, respectively. Overall, the results indicate that CodeVisionary consistently maintains effectiveness across tasks with varying difficulty levels, showcasing its adaptability and scalability.

D. Threats and Limitations

One threat to validity is that our benchmark may not cover all coding scenarios and programming languages. Hence, the experimental results may not be fully generalizable. In the future, we intend to extend our benchmark to include more. Another threat arises from the inherent randomness of LLMs. Since CodeVisionary relies on LLM agents to evaluate and score responses, the results may vary across trials. We therefore perform multiple trials and take the average as the experimental results.

VI. Related Work

A. Code Generation Evaluation

Current approaches for code generation evaluation can be categorized into metric-based, human-centered, and LLM-based. Early works directly transfer evaluation metrics from the natural language processing (NLP) domain to code generation, such as BLEU [7], ROUGE [30], and METEOR [31]. However, these metrics fail to capture the syntactic and functional correctness. To mitigate this limitation, CodeBLEU [8] incorporates syntactic and semantic information from Abstract Syntax Trees (AST) and Data Flow Graphs (DFG). Recently, execution-based metrics have gained prominence, including pass@k [9] and pass@t [10]. However, these metrics are based on high-quality unit tests and are limited to executable code.

For more complex and diverse tasks [32], [33], human-centered approaches become particularly important. Human evaluators can provide reliable and accurate evaluations based on their professional knowledge and programming experience, which are also essential for fine-tuning LLMs to better align with human preferences [34]. However, human-centered approaches are labour-intensive. Recently, LLM-based approaches [35] leverage LLMs to simulate the human evaluation process, enhancing efficiency and scalability. ICE-Score [14] aligns well with human preferences in terms of code accuracy and functionality without test oracles or references. CODEJUDGE [15] employs a "slow thinking" approach, enabling a thorough and reliable assessment of code's semantic correctness. Nevertheless, these LLM-based approaches still face limitations, including a lack of multi-dimensional context and omission of certain issues in complex code.

B. LLM-Based Agents

AI agents are artificial entities capable of accomplishing complex tasks by perceiving the environment and taking actions [36], [37]. Recently, rapid progress in LLMs has greatly increased researchers' attention to LLM-based agents [38], [39]. LLM-based agent frameworks leverage LLM as the central controller to address complex tasks by integrating external tools and powerful understanding capabilities of LLMs. These agents typically consist of four core components: planning, memory, perception, and action [36]. The planning component ensures efficient task execution through strategies such as single or multi-planner approaches [40], [41] and single or multi-turn planning [42], [43]. The memory component retains historical data to support reasoning, with implementations ranging from short-term to long-term memory [44], [45], as well as specific or shared memory [46], [47]. The perception component enables agents to process textual inputs [45], [48] and visual data [49], [50]. The action component integrates external tools [51], [52] for tasks such as web searches, file operations, and GUI interactions, thereby extending the agents' functional capabilities.

LLM-based agent frameworks designed for software engineering have demonstrated superior performance across various software development and maintenance tasks [53]–[57], including requirements engineering [17], [58], code generation [18], [59], static bug detection [19], [60], code review [61], [62], unit testing [63], system testing [45], [64], fault localization [65], [66], program repair [67], end-to-end software development [68], [69] and end-to-end software maintenance [70].

VII. Conclusion

This paper focuses on evaluating code generation for complex code scenarios and proposes a novel agent-based evaluation framework, named CodeVisionary. CodeVisionary consists of a requirement-guided multi-dimensional context distillation stage for gathering multi-dimensional contextual information based on the decomposed task requirements and

stepwise evaluation plan, and a fine-grained scoring and summarization stage for employing multiple judges engaging in discussions to better comprehend complex code in a fine-grained manner, finally reaching a consensus on the evaluation score. Besides, we provide detailed evaluation reports assisting developers in identifying shortcomings. Compared with the state-of-the-art approaches, the experimental results validate the effectiveness of CodeVisionary. In the future, we intend to further evaluate CodeVisionary on a broader range of datasets.

ACKNOWLEDGMENT

This research is supported by the National Natural Science Foundation of China under project (No. 62472126, 62276075), Natural Science Foundation of Guangdong Province (Project No. 2023A1515011959), and Shenzhen-Hong Kong Jointly Funded Project (Category A, No. SGDX20230116 091246007).

REFERENCES

[1] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[2] B. Shen, J. Zhang, T. Chen, D. Zan, B. Geng, A. Fu, M. Zeng, A. Yu, J. Ji, J. Zhao, Y. Guo, and Q. Wang, "Pangu-coder2: Boosting large language models for code with ranking feedback," *CoRR*, vol. abs/2307.14936, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2307.14936

[3] GitHub, "Github copilot - your ai pair programmer," https://github.com/features/copilot.

[4] A. Cursor, "Cursor," https://www.cursor.com/.

[5] M. Evtikhiev, E. Bogomolov, Y. Sokolov, and T. Bryksin, "Out of the BLEU: how should we assess quality of the code generation models?" *J. Syst. Softw.*, vol. 203, p. 111741, 2023. [Online]. Available: https://doi.org/10.1016/j.jss.2023.111741

[6] 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," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, ser. NIPS '23. Red Hook, NY, USA: Curran Associates Inc., 2023.

[7] K. Papineni, S. Roukos, T. Ward, and W. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*. ACL, 2002, pp. 311–318. [Online]. Available: https://aclanthology.org/P02-1040/

[8] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *CoRR*, vol. abs/2009.10297, 2020. [Online]. Available: https://arxiv.org/abs/2009.10297

[9] 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.

[10] T. X. Olausson, J. P. Inala, C. Wang, J. Gao, and A. Solar-Lezama, "Is self-repair a silver bullet for code generation?" in *International Conference on Learning Representations (ICLR)*, 2024.

[11] N. Rajkumar, R. Li, and D. Bahdanau, "Evaluating the text-to-sql capabilities of large language models," *CoRR*, vol. abs/2204.00498, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2204.00498

[12] L. Chen, Q. Guo, H. Jia, Z. Zeng, X. Wang, Y. Xu, J. Wu, Y. Wang, Q. Gao, J. Wang, W. Ye, and S. Zhang, "A survey on evaluating large language models in code generation tasks," *CoRR*, vol. abs/2408.16498, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2408.16498

[13] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *CoRR*, vol. abs/2406.00515, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2406.00515

[14] T. Y. Zhuo, "Ice-score: Instructing large language models to evaluate code," in *Findings of the Association for Computational Linguistics: EACL 2024, St. Julian's, Malta, March 17-22, 2024*, Y. Graham and M. Purver, Eds. Association for Computational Linguistics, 2024, pp. 2232–2242. [Online]. Available: https://aclanthology.org/2024.findings-eacl.148

[15] W. Tong and T. Zhang, "Codejudge: Evaluating code generation with large language models," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, Y. Al-Onaizan, M. Bansal, and Y. Chen, Eds. Association for Computational Linguistics, 2024, pp. 20 032–20 051. [Online]. Available: https://aclanthology.org/2024.emnlp-main.1118

[16] P. Wang, L. Li, L. Chen, Z. Cai, D. Zhu, B. Lin, Y. Cao, L. Kong, Q. Liu, T. Liu, and Z. Sui, "Large language models are not fair evaluators," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, L. Ku, A. Martins, and V. Srikumar, Eds. Association for Computational Linguistics, 2024, pp. 9440–9450. [Online]. Available: https://doi.org/10.18653/v1/2024.acl-long.511

[17] D. Jin, Z. Jin, X. Chen, and C. Wang, "MARE: multi-agents collaboration framework for requirements engineering," *CoRR*, vol. abs/2405.03256, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2405.03256

[18] D. Huang, Q. Bu, J. M. Zhang, M. Luck, and H. Cui, "Agentcoder: Multi-agent-based code generation with iterative testing and optimisation," *CoRR*, vol. abs/2312.13010, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2312.13010

[19] Z. Mao, J. Li, D. Jin, M. Li, and K. Tei, "Multi-role consensus through llms discussions for vulnerability detection," in *2024 IEEE 24th International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*. IEEE, 2024, pp. 1318–1319.

[20] L. Wang, W. Xu, Y. Lan, Z. Hu, Y. Lan, R. K. Lee, and E. Lim, "Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, A. Rogers, J. L. Boyd-Graber, and N. Okazaki, Eds. Association for Computational Linguistics, 2023, pp. 2609–2634. [Online]. Available: https://doi.org/10.18653/v1/2023.acl-long.147

[21] X. Jiang, Y. Dong, L. Wang, Z. Fang, Q. Shang, G. Li, Z. Jin, and W. Jiao, "Self-planning code generation with large language models," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 7, pp. 182:1–182:30, 2024. [Online]. Available: https://doi.org/10.1145/3672456

[22] prettier, "Prettier," https://github.com/prettier/prettier.

[23] jgm, "Pandoc," https://github.com/jgm/pandoc.

[24] J. Yang, J. Yang, K. Jin, Y. Miao, L. Zhang, L. Yang, Z. Cui, Y. Zhang, B. Hui, and J. Lin, "Evaluating and aligning codellms on human preference," *CoRR*, vol. abs/2412.05210, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2412.05210

[25] OpenAI, "GPT-3.5 Turbo fine-tuning and API updates," https://openai.com/index/gpt-3-5-turbo-fine-tuning-and-api-updates/.

[26] Anthropic, "Claude 3.5 Sonnet," https://www.anthropic.com/news/claude-3-5-sonnet.

[27] OpenAI, "Hello GPT-4o," https://openai.com/index/hello-gpt-4o/.

[28] scipy, "SciPy," https://github.com/scipy/scipy.

[29] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to mine aligned code and natural language pairs from stack overflow," in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 476–486.

[30] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Annual Meeting of the Association for Computational Linguistics*, 2004. [Online]. Available: https://api.semanticscholar.org/CorpusID:964287

[31] S. Banerjee and A. Lavie, "METEOR: an automatic metric for MT evaluation with improved correlation with human judgments," in *Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization@ACL 2005, Ann Arbor, Michigan, USA, June 29, 2005*, J. Goldstein, A. Lavie, C. Lin, and C. R. Voss, Eds. Association for Computational Linguistics, 2005, pp. 65–72. [Online]. Available: https://aclanthology.org/W05-0909/

[32] R. Bairi, A. Sonwane, A. Kanade, V. D. C., A. Iyer, S. Parthasarathy, S. K. Rajamani, B. Ashok, and S. Shet, "Codeplan: Repository-level

coding using llms and planning," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 675–698, 2024. [Online]. Available: https://doi.org/10.1145/3643757

[33] D. Shrivastava, D. Kocetkov, H. de Vries, D. Bahdanau, and T. Scholak, "Repofusion: Training code models to understand your repository," *CoRR*, vol. abs/2306.10998, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2306.10998

[34] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, ser. NIPS '22. Red Hook, NY, USA: Curran Associates Inc., 2022.

[35] R. Wang, J. Guo, C. Gao, G. Fan, C. Y. Chong, and X. Xia, "Can llms replace human evaluators? an empirical study of llm-as-a-judge in software engineering," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 1955–1977, 2025.

[36] Z. Xi, W. Chen, X. Guo, W. He, Y. Ding, B. Hong, M. Zhang, J. Wang, S. Jin, E. Zhou *et al.*, "The rise and potential of large language model based agents: A survey," *Science China Information Sciences*, vol. 68, no. 2, p. 121101, 2025.

[37] J. Liu, K. Wang, Y. Chen, X. Peng, Z. Chen, L. Zhang, and Y. Lou, "Large language model-based agents for software engineering: A survey," *CoRR*, vol. abs/2409.02977, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2409.02977

[38] W. Chen, Y. Su, J. Zuo, C. Yang, C. Yuan, C. Chan, H. Yu, Y. Lu, Y. Hung, C. Qian, Y. Qin, X. Cong, R. Xie, Z. Liu, M. Sun, and J. Zhou, "Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors," in *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. [Online]. Available: https://openreview.net/forum?id=EHg5GDnyq1

[39] S. Qiao, N. Zhang, R. Fang, Y. Luo, W. Zhou, Y. E. Jiang, C. Lv, and H. Chen, "Autoact: Automatic agent learning from scratch for QA via self-planning," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, L. Ku, A. Martins, and V. Srikumar, Eds. Association for Computational Linguistics, 2024, pp. 3003–3021. [Online]. Available: https://aclanthology.org/2024.acl-long.165

[40] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via chatgpt," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 7, pp. 189:1–189:38, 2024. [Online]. Available: https://doi.org/10.1145/3672459

[41] M. Josifoski, L. H. Klein, M. Peyrard, Y. Li, S. Geng, J. P. Schnitzler, Y. Yao, J. Wei, D. Paul, and R. West, "Flows: Building blocks of reasoning and collaborating AI," *CoRR*, vol. abs/2308.01285, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2308.01285

[42] E. Zelikman, Q. Huang, G. Poesia, N. D. Goodman, and N. Haber, "Parsel: Algorithmic reasoning with language models by composing decompositions," in *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., 2023. [Online]. Available: https://dl.acm.org/doi/10.5555/3666122.3667489

[43] A. Zhou, K. Yan, M. Shlapentokh-Rothman, H. Wang, and Y.-X. Wang, "Language agent tree search unifies reasoning, acting, and planning in language models," in *Proceedings of the 41st International Conference on Machine Learning*, ser. ICML'24. JMLR.org, 2024.

[44] M. Geva, R. Schuster, J. Berant, and O. Levy, "Transformer feed-forward layers are key-value memories," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, M. Moens, X. Huang, L. Specia, and S. W. Yih, Eds. Association for Computational Linguistics, 2021, pp. 5484–5495. [Online]. Available: https://doi.org/10.18653/v1/2021.emnlp-main.446

[45] J. Yoon, R. Feldt, and S. Yoo, "Autonomous large language model agents enabling intent-driven mobile GUI testing," *CoRR*, vol. abs/2311.08649, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2311.08649

[46] C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong, J. Xu, D. Li, Z. Liu, and M. Sun, "Chatdev: Communicative agents for software development," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, L. Ku, A. Martins, and V. Srikumar, Eds. Association for Computational Linguistics, 2024, pp. 15 174–15 186. [Online]. Available: https://doi.org/10.18653/v1/2024.acl-long.810

[47] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, and J. Schmidhuber, "Metagpt: Meta programming for A multi-agent collaborative framework," in *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. [Online]. Available: https://openreview.net/forum?id=VtmBAGCN7o

[48] J. A. Pizzorno and E. D. Berger, "Coverup: Coverage-guided llm-based test generation," *CoRR*, vol. abs/2403.16218, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2403.16218

[49] Z. Wang, W. Wang, Z. Li, L. Wang, C. Yi, X. Xu, L. Cao, H. Su, S. Chen, and J. Zhou, "Xuat-copilot: Multi-agent collaborative system for automated user acceptance testing with large language model," *CoRR*, vol. abs/2401.02705, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2401.02705

[50] M. Taeb, A. Swearngin, E. Schoop, R. Cheng, Y. Jiang, and J. Nichols, "Axnav: Replaying accessibility tests from natural language," in *Proceedings of the CHI Conference on Human Factors in Computing Systems, CHI 2024, Honolulu, HI, USA, May 11-16, 2024*, F. F. Mueller, P. Kyburz, J. R. Williamson, C. Sas, M. L. Wilson, P. O. T. Dugas, and I. Shklovski, Eds. ACM, 2024, pp. 962:1–962:16. [Online]. Available: https://doi.org/10.1145/3613904.3642777

[51] K. Zhang, G. Li, J. Li, Z. Li, and Z. Jin, "Toolcoder: Teach code generation models to use API search tools," *CoRR*, vol. abs/2305.04032, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.04032

[52] Z. Li, S. Dutta, and M. Naik, "Llm-assisted static analysis for detecting security vulnerabilities," *CoRR*, vol. abs/2405.17238, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2405.17238

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

[54] P. Gao, Z. Tian, X. Meng, X. Wang, R. Hu, Y. Xiao, Y. Liu, Z. Zhang, J. Chen, C. Gao *et al.*, "Trae agent: An llm-based agent for software engineering with test-time scaling," *arXiv preprint arXiv:2507.23370*, 2025.

[55] R. Hu, C. Peng, X. Wang, and C. Gao, "An llm-based agent for reliable docker environment configuration," *arXiv preprint arXiv:2502.13681*, 2025.

[56] X. Wang, P. Gao, X. Meng, C. Peng, R. Hu, Y. Lin, and C. Gao, "Aegis: An agent-based framework for bug reproduction from issue descriptions," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, ser. FSE Companion '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 331–342. [Online]. Available: https://doi.org/10.1145/3696630.3728557

[57] X.-C. Wen, J. Ye, C. Gao, L. Wu, and Q. Liao, "Evalsva: Multi-agent evaluators for next-gen software vulnerability assessment," *arXiv preprint arXiv:2501.14737*, 2024.

[58] C. Arora, J. Grundy, and M. Abdelrazek, "Advancing requirements engineering through generative ai: Assessing the role of llms," in *Generative AI for Effective Software Development*. Springer, 2024, pp. 129–148.

[59] Y. Ishibashi and Y. Nishimura, "Self-organized agents: A LLM multi-agent framework toward ultra large-scale code generation and optimization," *CoRR*, vol. abs/2404.02183, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2404.02183

[60] G. Fan, X. Xie, X. Zheng, Y. Liang, and P. Di, "Static code analysis in the AI era: An in-depth exploration of the concept, function, and potential of intelligent code analysis agents," *CoRR*, vol. abs/2310.08837, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2310.08837

[61] D. Tang, Z. Chen, K. Kim, Y. Song, H. Tian, S. Ezzini, Y. Huang, J. Klein, and T. F. Bissyandé, "Codeagent: Collaborative agents for software engineering," *CoRR*, vol. abs/2402.02172, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2402.02172

[62] Z. Rasheed, M. A. Sami, M. Waseem, K. Kemell, X. Wang, A. Nguyen, K. Systä, and P. Abrahamsson, "Ai-powered code review with llms: Early results," *CoRR*, vol. abs/2404.18496, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2404.18496

[63] Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen, and X. Peng, "No more manual tests? evaluating and improving chatgpt for unit

test generation," *CoRR*, vol. abs/2305.04207, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.04207

[64] G. Deng, Y. Liu, V. M. Vilches, P. Liu, Y. Li, Y. Xu, M. Pinzger, S. Rass, T. Zhang, and Y. Liu, "Pentestgpt: Evaluating and harnessing large language models for automated penetration testing," in *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, D. Balzarotti and W. Xu, Eds. USENIX Association, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/deng

[65] Y. Qin, S. Wang, Y. Lou, J. Dong, K. Wang, X. Li, and X. Mao, "Agentfl: Scaling llm-based fault localization to project-level context," *CoRR*, vol. abs/2403.16362, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2403.16362

[66] Z. Wang, Z. Liu, Y. Zhang, A. Zhong, J. Wang, F. Yin, L. Fan, L. Wu, and Q. Wen, "Rcagent: Cloud root cause analysis by autonomous agents with tool-augmented large language models," in *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, 2024, pp. 4966–4974.

[67] I. Bouzenia, P. T. Devanbu, and M. Pradel, "Repairagent: An autonomous, llm-based agent for program repair," *CoRR*, vol. abs/2403.17134, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2403.17134

[68] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," *Advances in Neural Information Processing Systems*, vol. 37, pp. 50 528–50 652, 2024.

[69] 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, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, M. Christakis and M. Pradel, Eds. ACM, 2024, pp. 1592–1604. [Online]. Available: https://doi.org/10.1145/3650212.3680384

[70] D. Chen, S. Lin, M. Zeng, D. Zan, J. Wang, A. Cheshkov, J. Sun, H. Yu, G. Dong, A. Aliev, J. Wang, X. Cheng, G. Liang, Y. Ma, P. Bian, T. Xie, and Q. Wang, "Coder: Issue resolving with multi-agent and task graphs," *CoRR*, vol. abs/2406.01304, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2406.01304