

# Agent-as-a-Judge: Evaluate Agents with Agents

Mingchen Zhuge<sup>1,2</sup>, Changsheng Zhao<sup>1</sup>, Dylan R. Ashley<sup>2</sup>, Wenyi Wang<sup>2</sup>, Dmitrii Khizbulin<sup>2</sup>, Yunyang Xiong<sup>1</sup>, Zechun Liu<sup>1</sup>, Ernie Chang<sup>1</sup>, Raghuraman Krishnamoorthi<sup>1</sup>, Yuandong Tian<sup>1</sup>, Yangyang Shi<sup>1</sup>, Vikas Chandra<sup>1</sup>, Jürgen Schmidhuber<sup>2</sup>

<sup>1</sup>Meta AI, <sup>2</sup>KAUST

Contemporary evaluation techniques are inadequate for agentic systems. These approaches either focus exclusively on final outcomes—ignoring the step-by-step nature of agentic systems, or require excessive manual labour. To address this, we introduce the **Agent-as-a-Judge** framework, wherein agentic systems are used to evaluate agentic systems. This is an organic extension of the LLM-as-a-Judge framework, incorporating agentic features that enable intermediate feedback for the entire task-solving process. We apply the Agent-as-a-Judge to the task of code generation. To overcome issues with existing benchmarks and provide a proof-of-concept testbed for Agent-as-a-Judge, we present **DevAI**, a new benchmark of 55 realistic automated AI development tasks. It includes rich manual annotations, like a total of 365 hierarchical user requirements. We benchmark three of the popular agentic systems using Agent-as-a-Judge and find it dramatically outperforms LLM-as-a-Judge and is as reliable as our human evaluation baseline. Altogether, we believe that Agent-as-a-Judge marks a concrete step forward for modern agentic systems—by providing rich and reliable reward signals necessary for dynamic and scalable self-improvement.

**Date:** October 18, 2024

**Correspondence:** [mingchen.zhuge@kaust.edu.sa](mailto:mingchen.zhuge@kaust.edu.sa), [cszhao@meta.com](mailto:cszhao@meta.com)

**Dataset:** <https://huggingface.co/devai-benchmark>

**Project:** <https://github.com/metauto-ai/agent-as-a-judge>

**Note:** First four authors made core contributions. KAUST crafted the dataset.

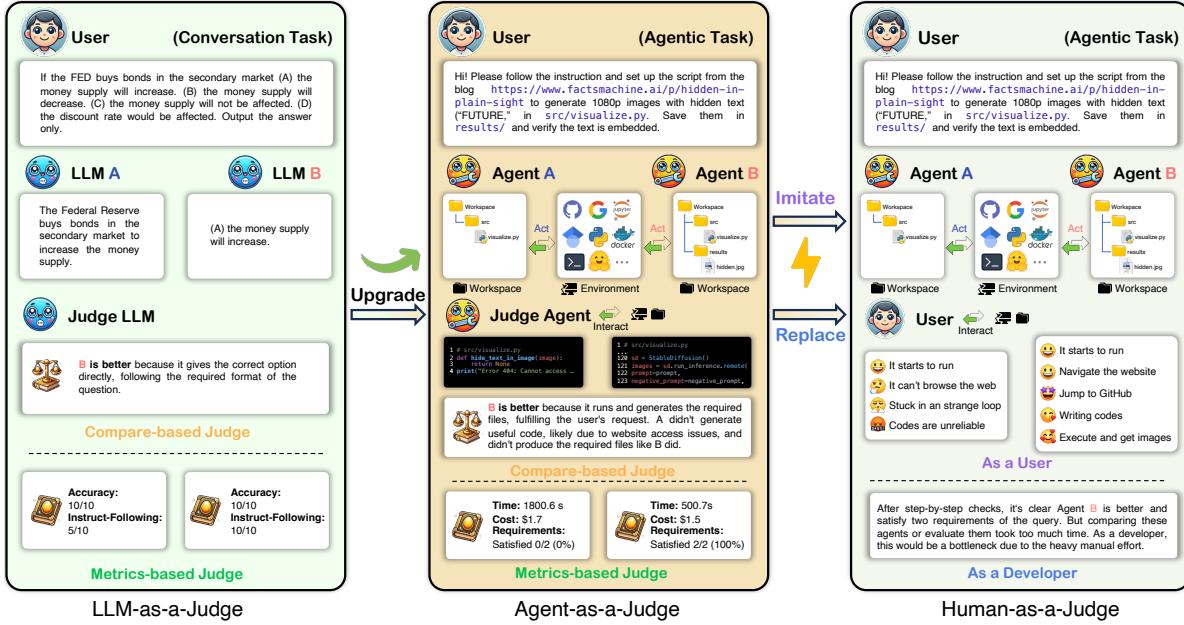
Work done while Mingchen was interning at Meta, with Changsheng leading.



## 1 Introduction

Recent years have seen multimodal agentic systems move from occasionally being able to solve small toy problems to being regularly deployed for challenging real-world problems (the dream of most AI research). Yet, the current evaluation methods and the available benchmarks for agentic systems are struggling to keep up with these rapid advances, dramatically slowing true progress.

We believe that the current issue with evaluating agentic systems stems from the lack of feedback during the intermediate task-solving stages for these nontraditional systems. Agentic systems think more like humans, often act step-by-step (Wooldridge, 1999) and often host very human-like symbolic communications internally to solve problems (Zhuge et al., 2023). And thus agentic systems should be evaluated like a human, with rich evaluative feedback which looks at the full thought and action trajectory; evaluating an agentic system in the traditional way is like evaluating a student using multiple-choice testing—a comparatively unreliable estimator (Park, 2010). For example, while SWE-Bench (Yang et al., 2024a) is widespread, its evaluation method, which relies solely on the final resolve rate for long-term automated repair tasks, does not effectively pinpoint what is happening within agentic systems that affects the resolve rate. On the other hand, performing a better evaluation with a human is prohibitively expensive. We instead propose that agentic systems should be used to evaluate agentic systems. Inspired by LLM-as-a-Judge (Zheng et al., 2024; Fu et al., 2023; Chen et al., 2024b), which uses LLMs to evaluate LLMs, we call this framework Agent-as-a-Judge, of which it is



**Figure 1** We introduce the Agent-as-a-Judge framework wherein agentic systems are used to evaluate agentic systems. We compare this to LLM-as-a-Judge, which uses LLMs to evaluate LLMs and for which Agent-as-a-Judge is a natural evolution, and Human-as-a-Judge, where skilled human labourers manually evaluate an agentic system.

a key extension to the world of agentic systems (see Figure 1). It not only retains the cost-effectiveness of LLM-as-a-Judge but is also equipped with agentic features, allowing it to provide rich intermediate feedback throughout the entire process, as it acts as an agentic system. We apply the Agent-as-a-Judge systems to the problem of evaluating code generating systems—one of the areas where agentic systems have looked the most promising recently.

In code generation, the development of benchmarks has also lagged behind the rapid advancement of agentic systems. HumanEval (Chen et al., 2021), for example, focuses exclusively on algorithmic problems, while MBPP (Austin et al., 2021) deals with simple programming tasks. Although they are useful for evaluating the basic skills of foundation models, neither of these two reflects the most practical challenges developers face. As a step away from this, SWE-Bench (Jimenez et al., 2023) did introduce more realistic problems from GitHub, offering a fresh approach to evaluation, but still primarily focuses on automated repairs tasks development process. Concerningly, recent research shows that large language models (LLMs) can already solve over 27% of the tasks in SWE-Bench without needing of advanced agentic systems (Xia et al., 2024). Equally concerning, recent work has begun to introduce mechanisms designed specifically for the individual tasks in the SWE-Bench dataset, leading to a lack of real-world generalization and violating Goodhart’s law: “When a measure becomes a target, it ceases to be a good measure” (Goodhart, 1976).

To address the aforementioned issues with the current benchmarks in code generation, we introduce DevAI: the AI Developer Dataset, which contains 55 real-world comprehensive AI app development tasks created by expert annotators. We apply three leading open-source code-generating agentic frameworks to the tasks in DevAI: MetaGPT (Hong et al., 2024b), GPT-Pilot (Pythagora.io, 2023), and OpenHands (Wang et al., 2024d). We evaluate their performance using human judges (a painstaking process), LLM-as-a-Judge (Zheng et al., 2024), and our Agent-as-a-Judge framework.

Through human evaluation, we found that GPT-Pilot and OpenHands were each able to satisfy about 29% of the task requirements in DevAI, but only one full task—showing that DevAI presents a good level of challenge to current systems. When comparing our human judges with our automatic Agent-as-a-Judge framework, we found that Agent-as-a-Judge aligns more closely with the consensus of our human judges (90%) as compared to LLM-as-a-Judge (70%) in all cases tested. In addition, we find that it aligns more closely with this ensemble than the individual human evaluators do, suggesting that—not only is it suitable as a replacement for a human evaluator—but it could in fact be more useful than an average lone human evaluator.

In addition, considering the evaluation cost, Agent-as-a-Judge saves 97.72% of the time and 97.64% of the cost compared to involving three human experts.

In summary, the principal contributions of this work are:

- We release the DevAI dataset, which consists of 55 comprehensive AI development tasks with accompanying tags, individual hierarchical requirements, and individual preferences.
- We benchmark three top open-source code generation agentic frameworks in DevAI, providing a more comprehensive analysis than previous evaluations of them.
- We introduce the general Agent-as-a-Judge concept, allowing agentic systems a fair and rich evaluation without the traditional costs associated with human involvement.
- We demonstrate that an Agent-as-a-Judge outperforms an LLM-as-a-Judge and performs comparably to human evaluators in our proof-of-concept.



**Tips:** We provide a paper outline and the experimental design in Appendices A and B.

## 2 DevAI: A Dataset for Automated AI Development

In this section, we introduce our new DevAI benchmark. We then evaluate three state-of-the-art code-generating agentic systems on this benchmark in [Section 3](#) and present their basic statistics.

### 2.1 Motivation

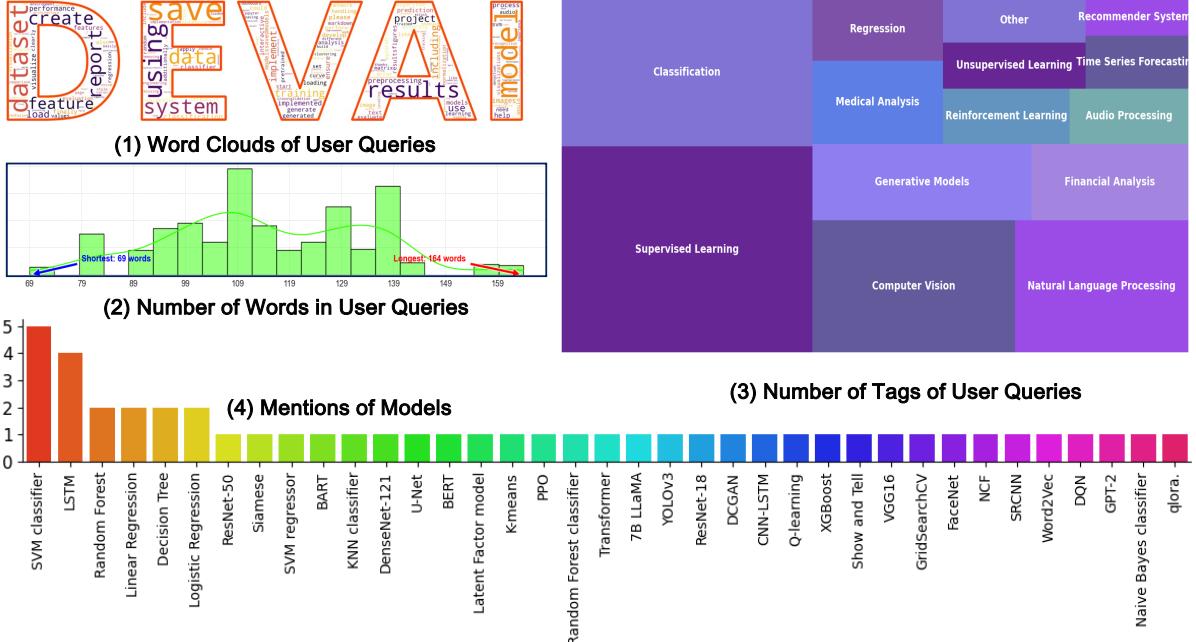
**Background** The code generation domain is an area where agentic systems have seen significant industrial deployment over the past two years (e.g., Devin<sup>1</sup> and Cursor<sup>2</sup>). However, in code generation, there isn't yet a benchmark that accurately reflects realistic user queries for developing complete AI systems. We believe this is because of the difficulty to evaluate such complex, real-world tasks. For example, while many companies advertise their systems based on its performance on benchmarks such as SWE-Bench ([Yang et al., 2024a](#)) (for automated repair) or HumanEval ([Chen et al., 2021](#)) (for algorithmic problems), these benchmarks cover only a small bit of an actual development process. Moreover, none of them accurately reflect the intermediate stages of development or provide sufficient reward signals for long-horizon development—similar issues are present in OpenAI's recent MLE-Bench ([Chan et al., 2024](#)). A benchmark that can evaluate the entire development process—ideally in a way that can help understand the degree to which current AI methods can reduce human labour—is missing.

**Topic** We chose automated AI development as our main topic. While AI and ML tasks are often more complex, they follow clear, standard procedures. For example, data processing typically comes first in an AI pipeline, and performance reporting goes at the end. We believe this topological nature can help better monitor the development process and provide useful signals to the agentic systems.

**Goals** An ideal benchmark should address critical issues in automated development by focusing on three key factors. First, it should reflect practical software scenarios, where tasks are often too complex for a single LLM, requiring human or agentic systems. Second, it should emphasize the development process, not just final outcomes (e.g., `pass@1` rates offer limited feedback and fail to highlight intermediate problems). Lastly, the evaluation should be computationally cost-effective and efficient, avoiding long training times or excessive manual oversight.

<sup>1</sup><https://www.cognition.ai/blog/introducing-devin>

<sup>2</sup><https://www.cursor.com/>



**Figure 2 Distribution of DevAI Tasks** (1) DevAI focuses on AI development tasks and so terms such as “dataset,” “model,” and “results” are particularly common in the queries. (2) The first 53 tasks in DevAI all have a one-paragraph query but of varying lengths (note that task 54 and 55 are excluded here as they are outliers, representing the longest and most complex tasks in the dataset). (3) Each task has one or more tags. The prevalence of supervised learning here reflects the fact that it dominates many machine learning applications. (4) SVM classifiers (Cortes, 1995) and LSTM models (Hochreiter, 1997) are two of the most widely used architectures—a fact reflected by DevAI.

## 2.2 The DevAI Dataset

Motivated by the ideas outlined above, we propose the DevAI dataset. DevAI consists of a curated set of 55 tasks, each defined by **(1)** a plain text user query that describes an AI development task; **(2)** a set of plain text requirements (for a total of 365 requirements), each with a set of dependencies connecting them to other requirements; and **(3)** a set of preferences (for a total of 125 preferences) which represent softer requirements.

DevAI is structured so that an agentic system starts by receiving a user query to begin development. The system is then evaluated on how well it meets the requirements, with preferences serving as optional, softer criteria. An example of one of the DevAI tasks can be seen in Figure 3.

The tasks in DevAI are relatively small-scale but cover commonly used key development techniques. As shown in Figure 2, our tasks are tagged and cover a variety of key areas in AI: supervised learning, reinforcement learning, computer vision, natural language processing, generative models, and others. Each of the tasks is a real-world problem that could be given to a research engineer, while simultaneously being relatively inexpensive computationally to run so as to reduce the cost of evaluating a method on this benchmark. Details of the sample collection and human labeling process for DevAI are provided in Appendix E.

The requirements belonging to each task represent a milestone in the comprehensive development process and are arranged as a directed acyclic graph (similar to the work by He et al. (2021)), with requirements such as visualizing results depending on correct data loading and modeling. This allows for more comprehensive non-sparse feedback than a binary success metric. Furthermore, the inclusion of hierarchical requirements makes simple memorization an inadequate solution strategy, as completing the entire task requires agentic capabilities rather than relying solely on symbolic memorization, as is typical in foundation models.

## 2.3 Preliminary Benchmark

We first conduct experiments to collect development outcomes from different frameworks, which serve as baselines in the DevAI dataset. We test three of the most popular open-source frameworks (which we

### Task 51: Devin\_AI\_Software\_Engineer\_Plants\_Secret\_Messages\_in\_Images

#### Query

*Hi! Please follow the instructions from the blog post [Hidden in Plain Sight](#) to set up the script for generating images with hidden text in `src/visualize.py`. Ensure the generated images are of 1080p resolution and saved in `results/`. Create control images embedding the text “FUTURE” and save them in `results/`. Please manually verify that the hidden text is embedded in the images.*

#### Requirements

##### ■ R0

**Criteria:** Follow the instructions from the blog post [Hidden in Plain Sight](#) to set up the script for generating images with hidden text in `src/visualize.py`.

**Dependencies** → {}

##### ■ R1

**Criteria:** Ensure the generated images are of 1080p resolution and saved in `results/`.

**Dependencies** → {R0}

##### ■ R2

**Criteria:** Create control images embedding the text “FUTURE” and save them in `results/`.

**Dependencies** → {R1}

#### Preferences (Optional)

##### ■ P0

**Criteria:** The system should be capable of learning and adapting to unfamiliar technologies and tools as required.

##### ■ P1

**Criteria:** After reviewing the blog post, ControlNet should successfully run on Modal to produce images with hidden messages for FUTURE.

**Figure 3 A task example in DevAI.** This task is adapted from a real-world demo given at <https://www.cognition.ai/blog/introducing-devin>. As this example shows, task requirements in DevAI are structured as a Directed Acyclic Graph (DAG), with nodes representing individual requirements and directed edges showing dependencies. More examples are in Appendix G.

will refer to as “**AI developers**”): MetaGPT (Hong et al., 2024b), GPT-Pilot (Pythagora.io, 2023), and OpenHands (Wang et al., 2024d)—all selected for their strong community acceptance (each having over 30,000 stars on GitHub).

**Experiment Setup** All of these three systems require a language model as a back-end engine, for which we use gpt-4o-2024-05-13, a state-of-the-art language model. These AI developers were given a time-limit of 1800 seconds to solve each task and were forcefully halted if they exceeded this time limit (we imposed this constraint, which was visible to the AI developers, as detailed in Appendix I). We capture the outputs generated during the automated development process, including code, files, and other artifacts. Additionally, we record key decisions and actions made by the agentic systems through some custom instrumentation code, resulting in a development trajectory for each of the agentic systems.

**Analysis** The basic statistics are shown in Table 1. MetaGPT is the most cost-efficient (1.19 USD), while OpenHands is the most expensive (6.38 USD). In terms of development time, OpenHands completes tasks

**Table 1 Preliminary Statistics of AI Developers.** We compare three leading open-source code agents using metrics such as average cost, average time, and the average number of generated files.

Metric	MetaGPT (Hong et al., 2024b)	GPT-Pilot (Pythagora.io, 2023)	OpenHands (Wang et al., 2024d)
 Basic Statistics			
Version	Data Interpreter (Hong et al., 2024a)	0.2.13	CodeAct v1.9 (Wang et al., 2024c)
(1) Average Cost	\$1.19	\$3.92	\$6.38
(2) Average Time	775.29s	1622.38s	362.41s
(3) Average Input Tokens	152863	606707	1252482
(4) Average Output Tokens	28546	59707	8457
(5) Average Saved Code Files	0.42	3.84	2.53
(6) Average Saved Code Lines	11.15	273.33	96.56
(7) Average Saved Files	4.42	5.91	3.60

in an average of 362.41s, while GPT-Pilot takes the longest at 1622.38s. On average, a full evaluation on DevAI with one of these three took around 210.65 USD and 14 hours to perform. While running, GPT-Pilot generates the most output tokens at 59707 tokens, whereas OpenHands processed the most at 1252482 tokens while producing the fewest at 8457 tokens. This suggests that OpenHands’s internal communication is more complicated but is more parsimonious in its decisions.

MetaGPT, while being the most cost-effective, generates fewer saved code files (0.42), suggesting it may be less inclined to save files. In contrast, GPT-Pilot generates the most saved files (3.84), reflecting a more prolific output. The difference in saved code lines, with GPT-Pilot saving 273.33 lines versus MetaGPT’s 11.15, underscores GPT-Pilot’s extensive output. Meanwhile, OpenHands, despite handling larger inputs, seems less focused on executing code to generate files, as evidenced by its lower file output (2.53 saved files). These statistics align with real user experiences (as discussed in Appendix F).

**Evaluations** Note that the results in Table 1 are not directly indicative of performance but provide valuable insights into the practical utility of DevAI and the performance of AI developers. The generated workspaces (generated files, code, etc.) and trajectories are utilized in subsequent experiments to perform evaluations using Human-as-a-Judge (section 3), LLM-as-a-Judge, and Agent-as-a-Judge (section 4).

### 3 Human-as-a-Judge: Manual Evaluation on DevAI

To determine the pragmatic validity of DevAI and to accurately estimate the actual code-generating abilities of current state-of-the-art agentic systems, in this section, we run and then manually evaluate the application of three AI developer baselines to DevAI. In Section 4, we show how this evaluation can be automated.

**Table 2 Human-as-a-Judge for AI Developers.** (I) and (D) represent independent performance versus performance considering task dependencies.  indicates multiple experts evolved, and  means the evaluations use white-box testing (allowing access to the generated workspace, human-collected trajectories, and open-source codebases). The results were derived from expert judgments and deliberations (see Appendix H).

Metric	MetaGPT (Hong et al., 2024b)	GPT-Pilot (Pythagora.io, 2023)	OpenHands (Wang et al., 2024d)
 /  Human-as-a-Judge			
(A) Requirements Met (I)	22.13%	44.80%	42.89%
(B) Requirements Met (D)	6.55%	28.96%	28.68%
(C) Self-Termination	41.81%	5.45%	54.54%
(D) Task Solve Rate	0.00%	1.81%	1.81%

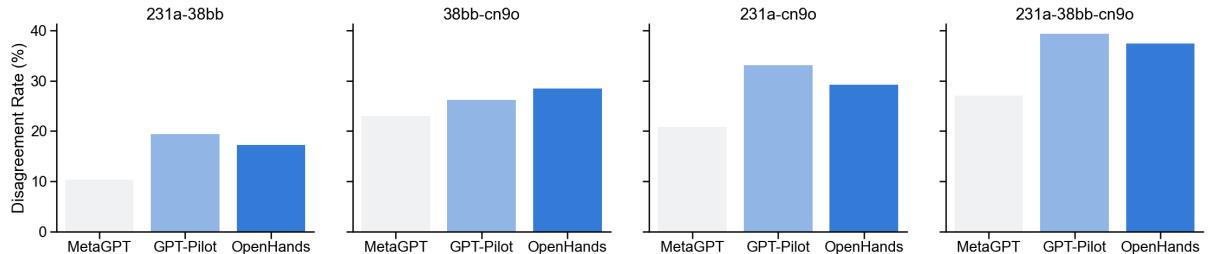
### 3.1 Benchmark Baselines by Human-as-a-Judge

**Human Evaluation Setup** After obtaining the baseline executions and conducting basic statistical analysis, we have three expert human evaluators (referred to here by their anonymous names: 231a, 38bb, and cn90) review the outputs of AI developer baselines to assess whether each requirement was satisfied. We have two rounds of human evaluations. To capture the bias inherent in typical human evaluation (this is desirable to capture here as it represents a likely scenario in deployment), in the first round, our evaluators first discussed the basic standards but were given minimal instructions. The templates the evaluators were given for the evaluation and their self-reported post-hoc descriptions of how they resolved ambiguities are reported in Figure 12 in Appendix H.

After the initial round of human evaluations (which totaled an estimated total of 58 human hours), we asked our evaluators to discuss and reach a consensus on their assessments (which took an estimated total of 28.5 additional human hours). This consensus, achieved after long sessions of debate, was used as the final human evaluation result for each method.

**Performance Analysis** The results of this experiment are shown in Table 2. We found that the two best-performing methods (GPT-Pilot and OpenHands) could satisfy about 29% of the requirements (or around 44% if prerequisites are ignored) but only on one task could they meet all the requirements. This highlights that DevAI offers a considerable but appropriate level of challenge for current and future methods. Moreover, the fulfillment of intermediate requirements aligns with our expectations, as discussed in Section 2, that DevAI provides richer feedback by uncovering how agentic systems falter during the process instead of just focusing on a single performance metric at the end.

### 3.2 Judging Human-as-a-Judge



**Figure 4** Between the three human evaluators, a large amount of disagreement was observed in their individual evaluations—highlighting the inherent unreliability of a single human evaluation.

**Disagreement Analysis** To analyze the presence of inductive bias and the reliability of the Human-as-a-Judge paradigm here, we calculate the disagreement rate between individual evaluators (shown in Figure 4). The results indicate that the disagreement rates between pairs of evaluators range from around 10% to 30%.

Due to the complexity of a complete AI development task, which typically involves multiple steps with varying outcomes at each step, humans can easily make errors when critical information is missed, such as environment feedback indicating small but severe coding errors or bugs. Additionally, some disagreements are not necessarily incorrect but arise from differing perspectives on how ambiguity should be resolved.

**Error Analysis** As previously noted, the evaluators engaged in a round of debating after their initial evaluations until they reached a consensus on each requirement in each task (with the results of this consensus evaluation shown in Table 2).

In our Human-as-a-Judge pipeline, evaluators could be convinced by evidence from others and acknowledge their judgment errors, adjusting their answers accordingly. This can be used to approximate individual errors. If the consensus evaluation more accurately predicts any extant ground truth, we would expect the majority vote from the individual evaluations to more closely approximate this than any single evaluation, due to the fundamental properties of ensemble classifiers (see Hastie et al. (2009)).

While the consensus evaluation may not represent the absolute ground truth (we acknowledge that some quantity of error likely would still exist even after this procedure), we expect the consensus evaluation to more accurately approximate any extant ground truth (Clemen, 1989). If this holds, the majority vote should align more closely with the consensus than with any individual evaluation. As shown in Figure 5, this is the case.

As seen in the results, although significant errors occur among all evaluators, the majority vote effectively corrects most of these errors. Notably, `cn9o` made the most errors (for example, 23.77% in evaluating GPT-Pilot). After applying the majority vote from all three evaluators, the overall error rate dropped to 6.01%, demonstrating the inherent benefits of majority voting.

**Conclusion** Human judgment errors are inevitable. To reduce them, we suggest two methods. First, like in this work, introduce a debate round after each judgment, where individuals present evidence and either persuade others or adjust their own opinions after discussion. This is particularly important when there are only a few evaluators, as majority voting with a small group can still lead to errors (around 5% compared to consensus evaluation, as shown in Figure 5). The second approach involves assembling a larger panel of experts (more is better when their accuracy exceeds 50% (Grofman et al., 1983)), with over 5 people recommended by [Hastie and Kameda \(2005\)](#); [Larrick and Soll \(2006\)](#), and relying on a majority vote. However, due to the high cost of engaging more experts and the fact that this is not always feasible in practice, we argue for the former.

Error Rate Comparison (%)			
	GPT-Pilot	MetaGPT Baseline	OpenHands
Labeler	12.57	7.92	10.93
	9.02	7.38	10.11
	23.77	16.67	21.86
	6.01	4.92	5.74
231a			
38bb			
<code>cn9o</code>			
majority_vote			

**Figure 5** Mismatch between the individual evaluations and the consensus evaluation. In particular, the majority vote classifier showed the smallest deviation from the consensus evaluation.

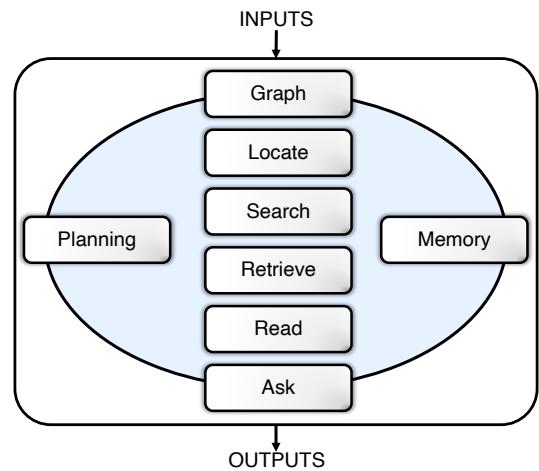
## 4 Agent-as-a-Judge: Evaluating Agents with Agents

Human evaluation, while somewhat reliable, is time-consuming and requires substantial expertise. To address this, we propose the Agent-as-a-Judge framework. If such an agentic system could evaluate like a human, it would reduce the need for human involvement and eliminate the trade-off between thoroughness and effort.

### 4.1 Proof-of-Concept

Based on our prior experiences with agent design and by imitating the human evaluation process, we initially designed eight modular, interacting components that form the foundation of our Proof-of-Concept for the Agent-as-a-Judge.

- (1) The **graph** module constructs a graph that captures the entire structure of the project, including files, modules, and dependencies. It can also break down chunks of code into code snippets.
- (2) The **locate** module identifies the specific folder or file referred to by a requirement.
- (3) The **read** module goes beyond simple file parsing, supporting the reading and understanding of multimodal data across 33 different formats, including code, images, videos and documents. This allows the agent to cross-reference various data streams and verify different kinds of requirement.
- (4) The **search** module provides a contextual understanding of code and can quickly retrieve highly relevant code snippets, as well as the nuances behind them (e.g., hidden dependencies).
- (5) The **retrieve** module



**Figure 6** Initial diagram of Agent-as-a-Judge.

**Table 3 AI Judges and Their Shift/Alignment with Human-as-a-Judge.** We compare the results of LLM-as-a-Judge and Agent-as-a-Judge with Human-as-a-Judge. (I) represents performance on independent tasks, while (D) represents performance considering task dependencies. Note:  gray-box settings use carefully manually collected trajectory data (which is nearly inaccessible in practical situations, see Appendix J). In contrast,  black-box setting doesn't need to access to such data. The red scores represent the absolute judge shift compared with Human-as-a-Judge (e.g., 2.74%).

Metric	MetaGPT (Hong et al., 2024b)	GPT-Pilot (Pythagora.io, 2023)	OpenHands (Wang et al., 2024d)
<input checked="" type="checkbox"/> LLM-as-a-Judge			
(a) Requirements Met (I)	19.39% (2.74%)	12.56% (32.24%)	11.47% (31.42%)
(b) Requirements Met (D)	1.63% (4.92%)	4.09% (24.87%)	2.18% (26.50%)
(c) Task Solve Rate	0.0% (0.0%)	0.0% (1.81%)	0.0% (1.81%)
<b>Alignment Rate ↑</b>	84.15%	65.30%	60.38%
<input checked="" type="checkbox"/> Agent-as-a-Judge			
(I) Requirements Met (I)	25.40% (3.26%)	53.00% (8.20%)	42.62% (0.27%)
(II) Requirements Met (D)	5.73% (0.81%)	39.89% (10.93%)	26.50% (2.17%)
(III) Task Solve Rate	0.0% (0.0%)	5.45% (3.64%)	1.81% (0.00%)
<b>Alignment Rate ↑</b>	88.52%	83.88%	90.44%
<input type="checkbox"/> LLM-as-a-Judge			
(a) Requirements Met (I)	28.68% (6.55%)	38.79% (4.10%)	43.16% (0.27%)
(b) Requirements Met (D)	17.75% (11.20%)	33.06% (4.10%)	32.24% (3.56%)
(c) Task Solve Rate	1.81% (1.81%)	3.63% (1.82%)	7.27% (5.46%)
<b>Alignment Rate ↑</b>	68.86%	71.85%	70.76%
<input type="checkbox"/> Agent-as-a-Judge			
(I) Requirements Met (I)	23.49% (1.35%)	46.44% (1.64%)	43.44% (0.54%)
(II) Requirements Met (D)	6.01% (0.54%)	30.60% (1.64%)	28.14% (0.53%)
(III) Task Solve Rate	0.0% (0.00%)	5.45% (3.64%)	3.63% (1.82%)
<b>Alignment Rate ↑</b>	92.07%	86.61%	90.16%
<input checked="" type="checkbox"/> Human-as-a-Judge			
<b>Alignment Rate</b> (38bb)	92.63%	90.98%	89.89%
<b>Alignment Rate</b> (cn9o)	83.33%	76.23%	78.15%
<b>Alignment Rate</b> (231a)	92.07%	87.43%	89.07%
Average of individuals	89.34%	84.88%	85.70%
Best of individuals	92.63%	90.98%	89.89%
<b>Alignment Rate</b> (Majority Vote)	95.08%	93.98%	94.26%

extracts information from long texts, identifying relevant segments in trajectories. With context from the above, (6) the `ask` module determines whether a given requirement is satisfied. (7) The `memory` module stores historical judgment information, allowing the agent to build on past evaluations. Finally, (8) the `planning` module plans the following actions, allowing the agent to strategize and sequence tasks based on the current state and the project goals.

Our initial design of the Agent-as-a-Judge, including all its components, is shown in Figure 6, and the operational process of the Agent-as-a-Judge is illustrated in Figure 9.

After conducting comprehensive ablation studies, we found that the modular combination of (1), (2), (3), (5), and (6) achieved the highest performance (see Appendix K). A sample of the dynamic evidence collected by the Agent-as-a-Judge is shown in Appendix M. We hypothesize this is because Agent-as-a-Judge needs high-quality factual information and is sensitive to noise. For example, while our design of the planning module introduces promising decision-making for future actions, the procedure is unstable. Initially, we hoped that historical information from the memory module would help to assess current requirements. However, it

proved detrimental, as any errors in previous judgments could lead to a chain of errors, negatively affecting current decisions. Besides, the current workspaces generated by developer agents, having only hundreds of lines of code, cannot fully benefit from the search module. The details of these findings are explained in Appendix K. Note that a perfect Agent-as-a-Judge is not the focus of this proof of concept, and thus, we leave the utilization of advanced agentic optimization methods for Agent-as-a-Judge, such as automated prompt optimization and workflow design (Zhuge et al., 2024; Hu et al., 2024), for future work.

## 4.2 Judging Agent-as-a-Judge and LLM-as-a-Judge

**Judge Shift** Judge Shift measures deviation from the Human-as-a-Judge consensus results, with lower values indicating a closer alignment. As shown in table 3, Agent-as-a-Judge consistently outperforms LLM-as-a-Judge across tasks, particularly those with task dependencies. For example, in Requirement (I), Agent-as-a-Judge shows a Judge Shift as low as 0.27%, while LLM-as-a-Judge reaches 31.24% for OpenHands. This underscores Agent-as-a-Judge’s stability and suitability for meeting task requirements. Furthermore, in the gray-box setting, both Agent-as-a-Judge and LLM-as-a-Judge show even better results than their performance in the black-box setting.

**Alignment Rate** The Alignment Rate reflects how closely the AI Judges’ evaluations align with human consensus across all 365 requirements. It is defined as the percentage of requirement evaluations that are the same as the Human-as-a-Judge consensus evaluation. Compared to LLM-as-a-Judge, Agent-as-a-Judge consistently achieves a higher Alignment Rate, closely matching human judgments. For example, when evaluating OpenHands, Agent-as-a-Judge reaches 92.07% and 90.44%, surpassing LLM-as-a-Judge’s 70.76% and 60.38% in both gray-box and black-box settings. This shows that Agent-as-a-Judge produces more accurate and human-aligned evaluations, especially in complex scenarios.

**PR Curves** Judging developer agents is a class-imbalanced task, where meeting requirements is much rarer than failing. Metrics like judge shift and alignment rate can be misleading. For example, since MetaGPT rarely meets requirements, LLM-as-a-Judge easily identifies most cases as negative (achieving 84.15% in the black-box setting). PR Curves offer a clearer performance measure by balancing precision and recall. Agent-as-a-Judge even outperforms any single human evaluator on OpenHands and aligns closest with majority voting. This shows that, in some cases, Agent-as-a-Judge can nearly replace human evaluators.

## 4.3 Ablations For Agent-as-a-Judge

We conduct ablations to evaluate the impact of adding different components on Agent-as-a-Judge’s performance. The components analyzed include `ask`, `graph`, `read`, `locate`, and `retrieve`. The component ablation study for Agent-as-a-Judge reveals key insights into the performance gains from adding specific functionalities.

With only the `ask` component, the agent achieves a 65.03% alignment rate. Adding the `graph` component increases performance to 75.95%, as the agent can better understand the relationships between files.

The introduction of `read` further improves the alignment rate to 82.24%, reflecting the value of direct access to the contents of the file. Incorporating `locate` brings a substantial boost to 90.44%, as the agent can efficiently target files relevant

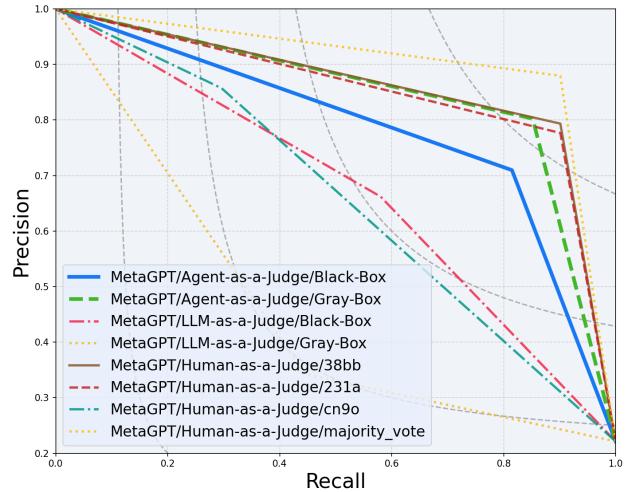


Figure 7 PR Curves comparing judge Methods.

**Table 4 Component Ablation Studies for Agent-as-a-Judge.** We analyze the impact of adding various components (`ask`, `graph`, `read`, `locate`, and `retrieve`) on the performance of Agent-as-a-Judge for judging OpenHands.

Metric	+ ask	+ graph	+ read	+ locate	+ retrieve
<b>Agent-as-a-Judge Performance</b>					
Alignment Rate					
Alignment Rate	65.03%	75.95%	82.24%	90.44%	90.16%

to the requirements. However, adding `retrieve` does not provide a significant benefit in this case. In contrast, as shown in [Table 3](#), the judgment of MetaGPT and GPT-Pilot indicates that `retrieve` is useful, as the trajectory provides additional valuable information.

#### 4.4 Cost Analysis

The Human-as-a-Judge took the three evaluators a self-reported total of 86.5 hours. With a 15 USD minimum wage (assuming this would buy a subject expert in AI), a full evaluation under DevAI would cost around 1297.50 USD. In comparison, Agent-as-a-Judge cost only 30.58 USD in API calls and took only 118.43 minutes—2.29% of the cost and 2.36% of the time of Human-as-a-Judge. LLM-as-a-Judge was faster at 10.99 minutes, but due to the absence of intelligent context selection by the Agent-as-a-Judge’s modules, it still cost 29.63 USD.

### 5 Related Work

Agentic systems and their applications are highly active research areas with numerous recent works having a relation to this work. This section details those works most relevant to ours. We provide a treatment of the less relevant related works in [Appendix D](#).

**AI Developers** AI in software development is growing fast ([Liu et al., 2024](#)). AI-driven developers have been applied to directly imitate software companies ([Hong et al., 2024b; Qian et al., 2024a](#)), debug code ([Yang et al., 2024a](#)), run data science methods ([Guo et al., 2024; Hong et al., 2024a; Li et al., 2024; Qiao et al., 2023](#)), and even write academic papers ([Lu et al., 2024a](#)).

**Benchmarks for AI developments** Benchmarks like MLAGentBench ([Huang et al., 2024](#)), ML-Bench ([Liu et al., 2023d](#)), SUPER ([Bogin et al., 2024](#)), DS-bench ([Jing et al., 2024](#)), and MLE-Bench ([Chan et al., 2024](#)) all focus on benchmarking agentic systems using AI tasks. However, DevAI distinguishes itself from all of these by focusing on realistic user queries that target a complete development cycle. It further includes a more comprehensive evaluation with multiple hierarchical requirements and preferences for each task. Comparatively, MLAGentBench ([Huang et al., 2024](#)) for example, focuses on final performance for a limited set of well-known tasks, which risks overfitting and fails to assess a system’s generalization or adaptability.

**AI Judges** Several works have looked at using AI systems as judges<sup>3</sup>. The work by [Chan et al. \(2023\); Zhao et al. \(2024\)](#), for example, extends LLM-as-a-Judge to have multiple LLMs in their evaluation process for conversational tasks. Unlike Agent-as-a-Judge, they employ a trivial agentic system and apply it only to evaluate LLMs under traditional evaluation setups. In contrast, ([Lu et al., 2024b](#)) uses a single LLM-based evaluator but, unlike LLM-as-a-Judge, applies this to multimodal tasks rather than just for evaluating LLMs. Less relevant are frameworks like those by [Chen et al. \(2024a\); Arora et al. \(2024\); Mündler et al. \(2024\)](#), where intermediate signals are used during coding development.

### 6 Discussion and Conclusion

**Outlook 1: Intermediate Feedback for Agentic Self-Improvement** A key power of the Agent-as-a-Judge, though not fully exploited here but nonetheless clear, is that it provides intermediate feedback that is essential for effective and efficient optimization ([Zhuge et al., 2024; Pan et al., 2024](#)). For example, [Agarwal et al. \(2019\)](#) proposes to solve the sparse reward problem in reinforcement learning, by learning auxiliary reward functions that provide intermediate feedback. Perhaps the greatest strength of the Agent-as-a-Judge framework is that an agentic system can use it to identify and fix issues in its solutions to complex, multistage problems on the fly—something older, delayed-feedback methods did not permit. By introducing Agent-as-a-Judge, we create the opportunity to build a process-supervised reward model (PRM) for improving agentic systems ([Lightman et al., 2023](#)).

---

<sup>3</sup>Additionally, we were pleased to find that a recent industry blog (<https://www.cognition.ai/blog/evaluating-coding-agents>), published 3 weeks before our submission, shares very similar ideas and provides further evidence that the Agent-as-a-Judge could have practical applications in agent systems.

**Outlook 2: Flywheel Effect Driven by Agent-as-a-Judge** The cycle of mutual improvement between the Agent-as-a-Judge and the evaluated agents, where both evolve together through iterative feedback, presents a promising direction. We hypothesize that an agentic version of a self-play system (Zelikman et al., 2022; Chen et al., 2024e; Wang et al., 2024b), could viably emerge by using the Agent-as-a-Judge as a key mechanism. Furthermore, the ongoing interaction between the Agent-as-a-Judge and the evaluated agents has the potential to create a flywheel effect, where successive incremental improvements reinforce one another, leading to progressively greater optimization and enhanced performance over time (Wang et al., 2022). This iterative process may also serve as a valuable complement to LLM reasoning data, help embedding agentic capabilities into foundation models (Luo et al., 2024).

**Conclusion** In this work, we introduced the Agent-as-a-Judge method to use agentic systems to evaluate agentic systems. We simultaneously released DevAI: a new benchmark that evaluates the code-generating ability of agentic systems on complete AI development tasks when used with Agent-as-a-Judge. We went on to show that Agent-as-a-Judge outperforms existing methods on this task and that it performs similarly to an ensemble of expert human evaluators. Altogether, we believe that the above opens the door for scaling up agentic far more than before.

## Acknowledgements

The authors thank Haozhe Liu, Piotr Piekos, Firas Laakom, Matteo Paltenghi for their suggestions or paper review. The research reported in this publication was supported by funding from the King Abdullah University of Science and Technology (KAUST) - Center of Excellence for Generative AI under award number 5940 and the SDAIA-KAUST Center of Excellence in Data Science and Artificial Intelligence.

## References

- Rishabh Agarwal, Chen Liang, Dale Schuurmans, and Mohammad Norouzi. Learning to generalize from sparse and underspecified rewards. In *International conference on machine learning*, pages 130–140. PMLR, 2019.
- Daman Arora, Atharv Sonwane, Nalin Wadhwa, Abhay Mehrotra, Saiteja Utpala, Ramakrishna Bairi, Aditya Kanade, and Nagarajan Natarajan. Masai: Modular architecture for software-engineering ai agents. *arXiv preprint arXiv:2406.11638*, 2024.
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.
- Anna Bavaresco, Raffaella Bernardi, Leonardo Bertolazzi, Desmond Elliott, Raquel Fernández, Albert Gatt, Esam Ghaleb, Mario Giulianelli, Michael Hanna, Alexander Koller, et al. Llms instead of human judges? a large scale empirical study across 20 nlp evaluation tasks. *arXiv preprint arXiv:2406.18403*, 2024.
- Ben Beglin, Kejuan Yang, Shashank Gupta, Kyle Richardson, Erin Bransom, Peter Clark, Ashish Sabharwal, and Tushar Khot. Super: Evaluating agents on setting up and executing tasks from research repositories. *arXiv preprint arXiv:2409.07440*, 2024.
- Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. Multipl-e: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691, 2023.
- Federico Cassano, Luisa Li, Akul Sethi, Noah Shinn, Abby Brennan-Jones, Jacob Ginesin, Edward Berman, George Chakhnashvili, Anton Lozhkov, Carolyn Jane Anderson, and Arjun Guha. Can it edit? evaluating the ability of large language models to follow code editing instructions, 2024. URL <https://arxiv.org/abs/2312.12450>.
- Chi-Min Chan, Weize Chen, Yusheng Su, Jianxuan Yu, Wei Xue, Shanghang Zhang, Jie Fu, and Zhiyuan Liu. Chateval: Towards better llm-based evaluators through multi-agent debate. *arXiv preprint arXiv:2308.07201*, 2023.
- Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal A. Patwardhan, Lilian Weng, and Aleksander Mkadry. Mle-bench: Evaluating machine learning agents on machine learning engineering. 2024. URL <https://api.semanticscholar.org/CorpusID:273233550>.

Harrison Chase. LangChain. <https://github.com/hwchase17/langchain>, 2022.

Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, et al. Coder: Issue resolving with multi-agent and task graphs. *arXiv preprint arXiv:2406.01304*, 2024a.

Dongping Chen, Ruoxi Chen, Shilin Zhang, Yinuo Liu, Yaochen Wang, Huichi Zhou, Qihui Zhang, Pan Zhou, Yao Wan, and Lichao Sun. Mllm-as-a-judge: Assessing multimodal llm-as-a-judge with vision-language benchmark. *arXiv preprint arXiv:2402.04788*, 2024b.

Dongping Chen, Yue Huang, Siyuan Wu, Jingyu Tang, Liuyi Chen, Yilin Bai, Zhigang He, Chenlong Wang, Huichi Zhou, Yiqiang Li, et al. Gui-world: A dataset for gui-oriented multimodal llm-based agents. *arXiv preprint arXiv:2406.10819*, 2024c.

Guiming Hardy Chen, Shunian Chen, Ziche Liu, Feng Jiang, and Benyou Wang. Humans or llms as the judge? a study on judgement biases. *arXiv preprint arXiv:2402.10669*, 2024d.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harry Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Zixiang Chen, Yihe Deng, Huizhuo Yuan, Kaixuan Ji, and Quanquan Gu. Self-play fine-tuning converts weak language models to strong language models. *arXiv preprint arXiv:2401.01335*, 2024e.

Ching-An Cheng, Allen Nie, and Adith Swaminathan. Trace is the new autodiff–unlocking efficient optimization of computational workflows. *arXiv preprint arXiv:2406.16218*, 2024.

Robert T Clemen. Combining forecasts: A review and annotated bibliography. *International journal of forecasting*, 5(4):559–583, 1989.

Corinna Cortes. Support-vector networks. *Machine Learning*, 1995.

Yijiang River Dong, Tiancheng Hu, and Nigel Collier. Can llm be a personalized judge? *arXiv preprint arXiv:2406.11657*, 2024.

Zhuoyun Du, Chen Qian, Wei Liu, Zihao Xie, Yifei Wang, Yufan Dang, Weize Chen, and Cheng Yang. Multi-agent software development through cross-team collaboration. *arXiv preprint arXiv:2406.08979*, 2024.

Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery in databases. *AI magazine*, 17(3):37–37, 1996.

Jinlan Fu, See-Kiong Ng, Zhengbao Jiang, and Pengfei Liu. Gptscore: Evaluate as you desire. *arXiv preprint arXiv:2302.04166*, 2023.

Charles Goodhart. *Monetary relationships: a view from Threadneedle Street*. University of Warwick, 1976.

Significant Gravitas. Auto-gpt. *GitHub repository*, 2023.

Bernard Grofman, Guillermo Owen, and Scott L Feld. Thirteen theorems in search of the truth. *Theory and decision*, 15(3):261–278, 1983.

Siyuan Guo, Cheng Deng, Ying Wen, Hechang Chen, Yi Chang, and Jun Wang. Ds-agent: Automated data science by empowering large language models with case-based reasoning. *arXiv preprint arXiv:2402.17453*, 2024.

Md Mahim Anjum Haque. *Fixeval: Execution-based evaluation of program fixes for competitive programming problems*. PhD thesis, Virginia Tech, 2023.

Reid Hastie and Tatsuya Kameda. The robust beauty of majority rules in group decisions. *Psychological review*, 112(2):494, 2005.

Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2 edition, 2009. doi: 10.1007/978-0-387-84858-7.

Xin He, Kaiyong Zhao, and Xiaowen Chu. Automl: A survey of the state-of-the-art. *Knowledge-based systems*, 212:106622, 2021.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

S Hochreiter. Long short-term memory. *Neural Computation MIT-Press*, 1997.

Sirui Hong, Yizhang Lin, Bangbang Liu, Biniao Wu, Danyang Li, Jiaqi Chen, Jiayi Zhang, Jinlin Wang, Lingyao Zhang, Mingchen Zhuge, et al. Data interpreter: An llm agent for data science. *arXiv preprint arXiv:2402.18679*, 2024a.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2024b.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024.

Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. Mlagentbench: Evaluating language agents on machine learning experimentation. In *Forty-first International Conference on Machine Learning*, 2024.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Peter Jansen, Marc-Alexandre Côté, Tushar Khot, Erin Bransom, Bhavana Dalvi Mishra, Bodhisattwa Prasad Majumder, Oyvind Tafjord, and Peter Clark. Discoveryworld: A virtual environment for developing and evaluating automated scientific discovery agents. *arXiv preprint arXiv:2406.06769*, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*, 2024.

Liqiang Jing, Zhehui Huang, Xiaoyang Wang, Wenlin Yao, Wenhao Yu, Kaixin Ma, Hongming Zhang, Xinya Du, and Dong Yu. Dsbench: How far are data science agents to becoming data science experts? *arXiv preprint arXiv:2409.07703*, 2024.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Chong Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Ruslan Salakhutdinov, and Daniel Fried. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. *arXiv preprint arXiv:2401.13649*, 2024.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR, 2023.

langchain ai. LangGraph. <https://github.com/langchain-ai/langgraph>, 2024.

Richard P Larrick and Jack B Soll. Intuitions about combining opinions: Misappreciation of the averaging principle. *Management science*, 52(1):111–127, 2006.

V Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Proceedings of the Soviet physics doklady*, 1966.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbulin, and Bernard Ghanem. Camel: Communicative agents for “mind” exploration of large scale language model society. *arXiv preprint arXiv:2303.17760*, 2023.

Ruochen Li, Teerth Patel, Qingyun Wang, and Xinya Du. Mr-copilot: Autonomous machine learning research based on large language models agents. *arXiv preprint arXiv:2408.14033*, 2024.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378 (6624):1092–1097, 2022.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let’s verify step by step. *arXiv preprint arXiv:2305.20050*, 2023.

- Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977*, 2024.
- Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*, 2023a.
- Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023b.
- Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chengguang Zhu. G-eval: Nlg evaluation using gpt-4 with better human alignment. *arXiv preprint arXiv:2303.16634*, 2023c.
- Yuliang Liu, Xiangru Tang, Zefan Cai, Junjie Lu, Yichi Zhang, Yanjun Shao, Zexuan Deng, Helan Hu, Zengxian Yang, Kaikai An, et al. Ml-bench: Large language models leverage open-source libraries for machine learning tasks. *arXiv preprint arXiv:2311.09835*, 2023d.
- Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The ai scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*, 2024a.
- Yujie Lu, Xianjun Yang, Xiujun Li, Xin Eric Wang, and William Yang Wang. Llmscore: Unveiling the power of large language models in text-to-image synthesis evaluation. *Advances in Neural Information Processing Systems*, 36, 2024b.
- Haipeng Luo, Qingfeng Sun, Can Xu, Pu Zhao, Qingwei Lin, Jianguang Lou, Shifeng Chen, Yansong Tang, and Weizhu Chen. Arena learning: Build data flywheel for llms post-training via simulated chatbot arena. *arXiv preprint arXiv:2407.10627*, 2024.
- Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. Code agents are state of the art software testers. *arXiv preprint arXiv:2406.12952*, 2024.
- Jiayi Pan, Yichi Zhang, Nicholas Tomlin, Yifei Zhou, Sergey Levine, and Alane Suhr. Autonomous evaluation and refinement of digital agents. In *First Conference on Language Modeling*, 2024.
- Jooyong Park. Constructive multiple-choice testing system. *British Journal of Educational Technology*, 41(6):1054–1064, 2010. doi: <https://doi.org/10.1111/j.1467-8535.2010.01058.x>. URL <https://bera-journals.onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8535.2010.01058.x>.
- Huy Nhat Phan, Phong X Nguyen, and Nghi DQ Bui. Hyperagent: Generalist software engineering agents to solve coding tasks at scale. *arXiv preprint arXiv:2409.16299*, 2024.
- Pythagora.io. Gpt-pilot: Your ai copilot for software development. <https://github.com/Pythagora-io/gpt-pilot>, 2023. URL <https://github.com/Pythagora-io/gpt-pilot>. GitHub repository.
- Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186, 2024a.
- Chen Qian, Zihao Xie, Yifei Wang, Wei Liu, Yufan Dang, Zhuoyun Du, Weize Chen, Cheng Yang, Zhiyuan Liu, and Maosong Sun. Scaling large-language-model-based multi-agent collaboration. *arXiv preprint arXiv:2406.07155*, 2024b.
- Bo Qiao, Liqun Li, Xu Zhang, Shilin He, Yu Kang, Chaoyun Zhang, Fangkai Yang, Hang Dong, Jue Zhang, Lu Wang, et al. Taskweaver: A code-first agent framework. *arXiv preprint arXiv:2311.17541*, 2023.
- Vyas Raina, Adian Liusie, and Mark Gales. Is llm-as-a-judge robust? investigating universal adversarial attacks on zero-shot llm assessment. *arXiv preprint arXiv:2402.14016*, 2024.
- N Reimers. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.
- Stephen Robertson, Hugo Zaragoza, et al. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389, 2009.
- Lin Shi, Weicheng Ma, and Soroush Vosoughi. Judging the judges: A systematic investigation of position bias in pairwise comparative assessments by llms. *arXiv preprint arXiv:2406.07791*, 2024.
- Linxin Song, Jiale Liu, Jieyu Zhang, Shaokun Zhang, Ao Luo, Shijian Wang, Qingyun Wu, and Chi Wang. Adaptive in-conversation team building for language model agents. *arXiv preprint arXiv:2405.19425*, 2024.

- Weihsiao Tan, Ziluo Ding, Wentao Zhang, Boyu Li, Bohan Zhou, Jumperg Yue, Haochong Xia, Jiechuan Jiang, Longtao Zheng, Xinrun Xu, et al. Towards general computer control: A multimodal agent for red dead redemption ii as a case study. *arXiv preprint arXiv:2403.03186*, 2024.
- Wei Tao, Yucheng Zhou, Wenqiang Zhang, and Yu Cheng. Magis: Llm-based multi-agent framework for github issue resolution. *arXiv preprint arXiv:2403.17927*, 2024.
- Aman Singh Thakur, Kartik Choudhary, Venkat Srinik Ramayapally, Sankaran Vaidyanathan, and Dieuwke Hupkes. Judging the judges: Evaluating alignment and vulnerabilities in llms-as-judges. *arXiv preprint arXiv:2406.12624*, 2024.
- Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Zhiyuan Liu, and Maosong Sun. Debugbench: Evaluating debugging capability of large language models. *arXiv preprint arXiv:2401.04621*, 2024.
- Michele Tufano, Anisha Agarwal, Jinu Jang, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. Autodev: Automated ai-driven development. *arXiv preprint arXiv:2403.08299*, 2024.
- Junyang Wang, Haiyang Xu, Haitao Jia, Xi Zhang, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. Mobile-agent-v2: Mobile device operation assistant with effective navigation via multi-agent collaboration. *arXiv preprint arXiv:2406.01014*, 2024a.
- Tianlu Wang, Ilia Kulikov, Olga Golovneva, Ping Yu, Weizhe Yuan, Jane Dwivedi-Yu, Richard Yuanzhe Pang, Maryam Fazel-Zarandi, Jason Weston, and Xian Li. Self-taught evaluators. *arXiv preprint arXiv:2408.02666*, 2024b.
- Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. *arXiv preprint arXiv:2402.01030*, 2024c.
- Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Opendevin: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024d.
- Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. *arXiv preprint arXiv:2212.10560*, 2022.
- Rüdiger Wirth and Jochen Hipp. Crisp-dm: Towards a standard process model for data mining. In *Proceedings of the 4th international conference on the practical applications of knowledge discovery and data mining*, volume 1, pages 29–39. Manchester, 2000.
- Michael Wooldridge. Intelligent agents. *Multiagent systems: A modern approach to distributed artificial intelligence*, 1: 27–73, 1999.
- Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 2023.
- Yiran Wu, Tianwei Yue, Shaokun Zhang, Chi Wang, and Qingyun Wu. Stateflow: Enhancing llm task-solving through state-driven workflows. *arXiv preprint arXiv:2403.11322*, 2024a.
- Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. Os-copilot: Towards generalist computer agents with self-improvement. *arXiv preprint arXiv:2402.07456*, 2024b.
- Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.
- Chengxing Xie, Canyu Chen, Feiran Jia, Ziyu Ye, Kai Shu, Adel Bibi, Ziniu Hu, Philip Torr, Bernard Ghanem, and Guohao Li. Can large language model agents simulate human trust behaviors? *arXiv preprint arXiv:2402.04559*, 2024.
- Tianyi Xiong, Xiyao Wang, Dong Guo, Qinghao Ye, Haoqi Fan, Quanquan Gu, Heng Huang, and Chunyuan Li. Llava-critic: Learning to evaluate multimodal models. *arXiv preprint arXiv:2410.02712*, 2024.
- Tianqi Xu, Linyao Chen, Dai-Jie Wu, Yanjun Chen, Zecheng Zhang, Xiang Yao, Zhiqiang Xie, Yongchao Chen, Shilong Liu, Bochen Qian, et al. Crab: Cross-environment agent benchmark for multimodal language model agents. *arXiv preprint arXiv:2407.01511*, 2024.

- John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024a.
- Zhao Yang, Jiaxuan Liu, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. Appagent: Multimodal agents as smartphone users. *arXiv preprint arXiv:2312.13771*, 2023.
- Zhiyu Yang, Zihan Zhou, Shuo Wang, Xin Cong, Xu Han, Yukun Yan, Zhenghao Liu, Zhixing Tan, Pengyuan Liu, Dong Yu, et al. Matplotagent: Method and evaluation for llm-based agentic scientific data visualization. *arXiv preprint arXiv:2402.11453*, 2024b.
- Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488, 2022.
- Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*, 2023.
- Ruochen Zhao, Wenxuan Zhang, Yew Ken Chia, Deli Zhao, and Lidong Bing. Auto arena of llms: Automating llm evaluations with agent peer-battles and committee discussions. *arXiv preprint arXiv:2405.20267*, 2024.
- Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36, 2024.
- Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. Codebertscore: Evaluating code generation with pretrained models of code. *arXiv preprint arXiv:2302.05527*, 2023a.
- Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023b.
- Wangchunshu Zhou, Yuchen Eleanor Jiang, Long Li, Jialong Wu, Tiannan Wang, Shi Qiu, Jintian Zhang, Jing Chen, Ruipu Wu, Shuai Wang, et al. Agents: An open-source framework for autonomous language agents. *arXiv preprint arXiv:2309.07870*, 2023c.
- Wangchunshu Zhou, Yixin Ou, Shengwei Ding, Long Li, Jialong Wu, Tiannan Wang, Jiamin Chen, Shuai Wang, Xiaohua Xu, Ningyu Zhang, et al. Symbolic learning enables self-evolving agents. *arXiv preprint arXiv:2406.18532*, 2024.
- Mingchen Zhuge, Haozhe Liu, Francesco Faccio, Dylan R Ashley, Róbert Csordás, Anand Gopalakrishnan, Abdullah Hamdi, Hasan Abed Al Kader Hammoud, Vincent Herrmann, Kazuki Irie, et al. Mindstorms in natural language-based societies of mind. *arXiv preprint arXiv:2305.17066*, 2023.
- Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbulin, and Jurgen Schmidhuber. Language agents as optimizable graphs. *arXiv preprint arXiv:2402.16823*, 2024.
- Terry Yue Zhuo. Ice-score: Instructing large language models to evaluate code. *arXiv preprint arXiv:2304.14317*, 2023.
- Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

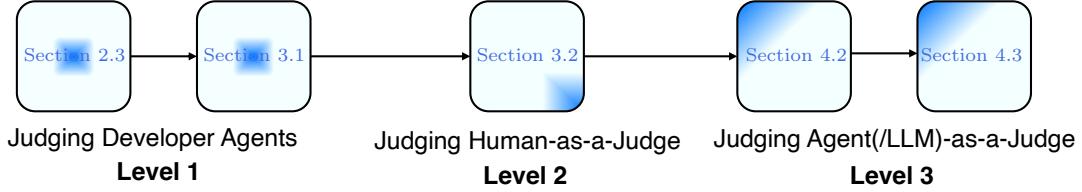
## A Outline of this Paper



Figure 8 We Outline the Logical Flow of the Agent-as-a-Judge Framework.

## B Experiment Designs

This section outlines the experimental designs aimed at evaluating developer agents' performance, analyzing human evaluations, and comparing AI-based judging systems. The experiments are structured across three levels, as illustrated below.



### B.1 Summary of Experiments

The experiments are categorized into three levels as follows:

#### Level 1

- ⌚ *Experiment 1a:* Basic performance statistics for developer agents ([Section 2.3](#))
- ⌚ *Experiment 1b:* Human evaluations of developer agents ([Section 3.1](#))

#### Level 2

- ⌚ *Experiment 2a:* Error analysis of human evaluations ([Section 3.2](#))

#### Level 3

- ⌚ *Experiment 3a:* AI judge baselines ([Section 4.2](#))
- ⌚ *Experiment 3b:* Ablation studies for Agent-as-a-Judge ([Section 4.3](#))

### B.2 Judges and Subjects of Evaluation

The following table summarizes the judge and the subject being evaluated in each experiment:

Experiment	Who is the Judge?	Who is being Judged?
Section 2.3	<i>Human</i>	<i>Developer Agents</i>
Section 3.1	<i>Human</i>	<i>Developer Agents</i>
Section 3.2	<i>Human</i>	<i>Human</i>
Section 4.2	(1) <i>LLM-as-a-Judge</i> (2) <i>Agent-as-a-Judge</i> (3) <i>Human</i> (4) <i>Human</i>	(1) <i>Developer Agents</i> (2) <i>Developer Agents</i> (3) <i>LLM-as-a-Judge</i> (4) <i>Agent-as-a-Judge</i>
Section 4.3	<i>Human</i>	<i>Agent-as-a-Judge</i>

## C Agent-as-a-Judge Pipeline



**Figure 9** The pipelines of developer agents and judge agent. Some materials in this figure are from original blog (<https://www.factsmachine.ai/p/hidden-in-plain-sight>).

## D Extend Related Work

Our main paper includes mostly related works of **AI developers**, **Benchmarks for AI developments**, and **AI judges**. However, the following works contribute significantly to the community and also relate to this work. We record this work as additional related work.

**LLM-based Autonomous Agents** Recent developments in LLM-based agents have expanded their capabilities beyond simple task execution to more autonomous problem-solving and decision-making. AutoGPT (Gravitas, 2023) and LangChain (Chase, 2022) provide frameworks for single-agent systems that leverage external tools for more complex tasks. Similarly, research such as CAMEL (Li et al., 2023), MetaGPT (Hong et al., 2024b), NLSOM (Zhuge et al., 2023), AutoGen (Wu et al., 2023) focus on role-based multi-agent communication, improving collaboration among agents. However, the challenge of maintaining coherence in agents’ dialogue and preventing hallucination remains prominent (Du et al., 2024; Zhou et al., 2023c). Besides, Agent-trust (Xie et al., 2024) examines if LLM agents, like GPT-4, can simulate human trust behaviors in Trust Games, showing that they can align with human behavior.

Most recently, using graphs to build agents has gained prominence. Earlier work like GPTSwarm (Zhuge et al., 2024) and LangGraph (langchain ai, 2024) proposed using nodes to represent operations and edges to represent the connections between them. In GPTSwarm, multiple agents represented as subgraphs in a graph are connected by optimizable edges, and reinforcement learning is employed to optimize the edges. Following this approach, several agent frameworks have incorporated graphs into their designs (Hong et al., 2024a; Zhou et al., 2024; Qian et al., 2024b). Additionally, various optimization methods have been developed to enhance agent performance further (Wu et al., 2024a; Song et al., 2024; Hu et al., 2024). In practical applications, many studies focus on understanding and interacting with GUIs (Wang et al., 2024a; Chen et al., 2024c; Yang et al., 2023; Xu et al., 2024; Tan et al., 2024; Wu et al., 2024b). For code generation agents (Jin et al., 2024), current research mainly emphasizes automated repair (Yang et al., 2024a; Phan et al., 2024; Tao et al., 2024), computational modular design (Khattab et al., 2023; Cheng et al., 2024), and automated development (Tufano et al., 2024; Huang et al., 2023). Among these, open-sourced frameworks like OpenHands (Wang et al., 2024d) have gained popularity due to their strong user experience. Moreover, scientific discovery (Jansen et al., 2024; Lu et al., 2024a) and ML agents (Yang et al., 2024b) are also receiving increased attention.

**LLM-as-a-Judge** In the domain of AI evaluation and judgment, frameworks (Zheng et al., 2024; Fu et al., 2023) have pioneered the use of LLMs to assess conversational agents, demonstrating how LLMs can evaluate dialogue quality and consistency. LLM-as-a-judge has also expanded into the multimodal domain, providing clear visual-language feedback (Chen et al., 2024b; Xiong et al., 2024). ICE-Score (Zhuo, 2023) improves on metrics like CodeBERTScore (Zhou et al., 2023a) and G-EVAL Liu et al. (2023c) by better aligning with human preferences and functional correctness. Expanding beyond dialogue, LLMs like CodeR (Chen et al., 2024a) and MASAI (Arora et al., 2024) apply similar judging principles to the code validation process, where AI systems autonomously evaluate and verify computer programs. Our work builds on these advancements by exploring how LLMs can perform more nuanced judgment tasks, further investigating their potential in decision-making across various domains. Recent research also focuses on judging LLM-as-a-Judges (Chen et al., 2024d; Bavaresco et al., 2024; Thakur et al., 2024; Dong et al., 2024; Shi et al., 2024; Raina et al., 2024).

**Coding Benchmarks** Recent advances in code generation have led to the innovation of various benchmarks to evaluate model performance (Liu et al., 2024). Early benchmarks, such as MBPP (Austin et al., 2021), HumanEval (Chen et al., 2021), and MultiPL-E (Cassano et al., 2023), focus primarily on generating simple functions. While these benchmarks are useful for evaluating the correctness of generated code, they are limited in complexity and do not fully represent the challenges encountered in real-world software development.

As the field progressed, newer benchmarks began to focus on more complex and realistic tasks. APPS (Hendrycks et al., 2021), CodeContests (Li et al., 2022), and LiveCodeBench (Jain et al., 2024) moved toward competitive programming challenges that involve advanced algorithms and data structures. These tasks are more representative of problems encountered in coding competitions and help push models toward more sophisticated problem-solving. DS-1000 (Lai et al., 2023) was introduced to assess the skills of models with data science libraries, evaluating their ability to use APIs and execute complex data analysis workflows.

Meanwhile, AgentBench (Liu et al., 2023b) focuses on testing reasoning and decision-making abilities in interactive environments, highlighting differences in performance between commercial and open-source models.

To address real-world programming needs beyond code generation, specialized benchmarks have been created to evaluate tasks such as debugging, refactoring, and code navigation. BigCodeBench (Zhao et al., 2024) has advanced the evaluation of LLMs by testing their ability to solve complex, real-world programming tasks beyond traditional algorithmic challenges. CANITEDIT (Cassano et al., 2024), DebugBench (Tian et al., 2024), and FixEval (Haque, 2023) evaluate the ability of a model to edit and improve existing code. Additionally, benchmarks such as SWE-Bench (Jimenez et al., 2023) focus on resolving issues in GitHub repositories, simulating practical software development scenarios. Finally, benchmarks such as RepoBench (Liu et al., 2023a) and RepoEval (Zhang et al., 2023) delve into the evaluation of models in large-scale, multifile codebases. These benchmarks measure the ability of language models to understand the structure of repositories and solve problems within more complex, collaborative environments.

There are also tasks that assess tool-use skills requiring programming knowledge, though not directly focused on programming. CRAB (Xu et al., 2024) sets a new standard for benchmarking autonomous agents by supporting cross-environment tasks and offering a more comprehensive evaluation framework. Similarly, WebArena-serious (Zhou et al., 2023b; Koh et al., 2024) introduces a realistic benchmark for evaluating multimodal agents on complex, real-world web tasks, addressing the limitations of text-only agents by integrating both visual and textual inputs across diverse domains.

## E The Procedures of Creating DevAI Dataset

### E.1 Manually draft user queries

Given the execution cost of the developer agents, we collect small-scale AI tasks to ensure the practical applicability of our benchmark. Since these tasks are small-scale and well studied, which are easy to overfit in terms of task performance metrics, unlike previous benchmarks (e.g., (Huang et al., 2024)), we do not evaluate task performance as the development performance measure. Instead, we prioritize the step-by-step task-solving ability, which is essential for real-world development. Our queries are specifically designed to require the development agents to understand user intentions, solve the task in multiple steps, and adapt to unexpected step outcomes. This approach also makes our benchmark user-friendly, transparent, and better reflects real-world deployment situations. To enable effective evaluation, our queries present a specific development file structure for the developer agents to follow. To ensure that the developer agents save the files to be evaluated in the workspace, we develop constraint prompts added to the query to form an extended query. The constraint prompts guide the developer agents to save and execute the source codes, which are in line with the real-world development standard. See Appendix I for our constraint prompts.

### E.2 Set Judging Criteria

To make the evaluation of the developer agent precise, we assign to each task query a list of requirements as task milestones. The requirements are chosen so that satisfying all the requirements is a necessary condition to consider the task to be solved successively. Since our tasks are AI-centric, our queries target essential elements of AI development, including data processing, AI method, presentation of evaluation metrics, visualization, and human-computer interaction, covering the key areas that matter most in real-world scenarios. On the other hand, breaking down tasks into individual requirements also reflects the multi-step nature of code development. Importantly, to avoid ambiguity, we set the requirements to be explicit, binary, and straightforward to evaluate. To include other human predispositions, we include a list of preferences per task that covers subjective, ambiguous, or non-explicitly stated characteristics.

### E.3 Building Dependency Among Requirements

To enhance the realism of our benchmark, we analyzed the dependencies among requirements. Over the past decades, methodologies such as the KDD Process (Fayyad et al., 1996) and CRISP-DM (Wirth and Hipp, 2000) have guided ML/AI development, establishing foundational frameworks that have been further refined with the rise of AutoML (He et al., 2021).

Inspired by these methodologies, we identified a seven-step process for automated AI development tasks. This process includes critical stages such as data preprocessing, feature engineering, model selection, and hyperparameter tuning, along with essential post-development activities like metrics recording, report generation, and the development of interactive user applications. This structured approach allows us to evaluate the agent's ability to manage task dependencies and effectively navigate complex, real-world scenarios.

### E.4 Refine the dataset

Manual refinements were necessary to ensure the accuracy and clarity of DevAI. We perform two rounds of comprehensive review and edits on DevAI, each round being done by a different participant. During these manual refinements, we focus on the logical consistency of our queries and requirements, the correctness and ambiguity of the language, and the applicability of the content to the task domain. We identified a moderate number of errors in our dataset during the review.

### E.5 Analyse the dataset

We categorized each requirement based on its focus, whether it was related to the data set, the machine learning method, visualization, metrics, HCI, or data processing. While this approach provides a useful framework, it is important to recognize that these categories might overlap or miss certain nuances. Similarly, preferences were classified by how strongly they appeared in the query text, ranging from those inferred by

common sense to those explicitly stated. Although this helps to organize preferences, it is worth noting that such classifications can be subjective and may not fully capture the importance of each preference in practical applications. By structuring the requirements and preferences this way, we aim to enhance the evaluation process, though flexibility and context awareness remain crucial for truly robust assessments.

## E.6 Auxiliary Information

Some of the tasks require the download of a Kaggle data set, where a Kaggle credential is needed. Our constraint [I](#) requires an “`is_kaggle_api_needed`” tag to determine whether the credential is needed to be included in the extended query. We further mark each task with some tags describing the AI sub-fields related to the task, including computer vision, supervised learning, reinforcement learning, natural language processing, etc., as well as two “`is_training_needed`” and “`is_web_navigation_needed`” tags as auxiliary information. We also categorize each requirement into one of the following: (1) `dataset or environment`, (2) `data preprocessing and postprocessing`, (3) `machine learning method`, (4) `save trained model`, (4) `performance metrics`, (5) `human computer interaction`, (6) `visualization`, and (7) `other`, reflecting the nature of the requirement.

## E.7 A Json Format of Our Sample

Here, we provide a sample of the DevAI with its json format. We also provide more samples in [Appendix G](#).

```
{
  "name": "25_Speech_Emotion_Recognition_CNN_LSTM_RAVDESS_DL",
  "query": "I am seeking a speech emotion recognition project using a CNN-LSTM model with the RAVDESS dataset, which should be downloaded from Kaggle or [this Hugging Face link](https://huggingface.co/datasets/xbgoose/ravdess). The project should load the dataset and perform robust audio preprocessing (noise removal and normalization) and MFCC feature extraction, implemented in `src/data_loader.py`. The CNN-LSTM model should be implemented in 'src/model.py'. Recognition accuracy should be saved in `results/metrics/recognition_accuracy.txt`, and a confusion matrix should be generated and saved as `results/figures/confusion_matrix.png`. Additionally, a user-friendly local API should be created using Flask to allow users to upload audio files and receive emotion recognition results, with the implementation included in `src/hci.py`.",
  "tags": [
    "Audio Processing",
    "Classification"
  ],
  "requirements": [
    {
      "requirement_id": 0,
      "prerequisites": [],
      "criteria": "The \"RAVDESS\" dataset is loaded in `src/data_loader.py`, which is downloaded from Kaggle or [this Hugging Face link](https://huggingface.co/datasets/xbgoose/ravdess).",
      "category": "Dataset or Environment",
      "satisfied": null
    },
    {
      "requirement_id": 1,
      "prerequisites": [
        0
      ],
      "criteria": "Audio preprocessing, including noise removal and normalization, is implemented in `src/data_loader.py`.",
      "category": "Data preprocessing and postprocessing",
      "satisfied": null
    }
  ]
}
```

```

{
    "requirement_id": 2,
    "prerequisites": [
        0,
        1
    ],
    "criteria": "MFCC feature extraction is implemented in `src/data_loader.py`.",
    "category": "Data preprocessing and postprocessing",
    "satisfied": null
},
{
    "requirement_id": 3,
    "prerequisites": [],
    "criteria": "The \"CNN-LSTM\" model is implemented in 'src/model.py'.",
    "category": "Machine Learning Method",
    "satisfied": null
},
{
    "requirement_id": 4,
    "prerequisites": [
        2,
        3
    ],
    "criteria": "Recognition accuracy is saved in `results/metrics/recognition_accuracy.txt`.",
    "category": "Performance Metrics",
    "satisfied": null
},
{
    "requirement_id": 5,
    "prerequisites": [
        2,
        3,
        4
    ],
    "criteria": "The confusion matrix is generated and saved as `results/figures/confusion_matrix.png`.",
    "category": "Visualization",
    "satisfied": null
},
{
    "requirement_id": 6,
    "prerequisites": [
        2,
        3
    ],
    "criteria": "A local API is created using \"Flask\" to allow users to upload audio files and receive emotion recognition results. The implementation should be included in `src/hci.py`.",
    "category": "Human Computer Interaction",
    "satisfied": null
}
],
"preferences": [
{
    "preference_id": 0,
    "criteria": "The audio preprocessing step should be robust, effectively reducing noise while preserving the integrity of the speech signals."
}
]

```

```
        "satisfied": null
    },
{
    "preference_id": 1,
    "criteria": "The local API should be user-friendly, with clear
                instructions for uploading files and interpreting results.",
    "satisfied": null
}
],
"is_kaggle_api_needed": true,
"is_training_needed": true,
"is_web_navigation_needed": true
}
```

---

## F User experiences of code-generation agentic systems

OpenHands ([Wang et al., 2024d](#)) offers the most refined user experience, leveraging its highly interactive frontend to enable seamless user interaction and task execution. This interface allows users to engage directly with the system, resulting in a smoother and more intuitive workflow, which drives operational efficiency.

In contrast, MetaGPT ([Hong et al., 2024b](#)) excels in task decomposition through its use of Directed Acyclic Graphs (DAGs), a well-structured and scalable approach aligned with industry best practices in system modularization. This enhances its appeal for users focused on task clarity and modular breakdowns. However, in practical deployments, MetaGPT tends to be less aggressive in file management and preservation, potentially due to its core positioning as a data analysis tool, which does not prioritize persistent state management. Similarly, OpenDevin demonstrates a notable overconfidence in its code generation, frequently skipping the critical step of post-generation code execution, requiring users to intervene manually.

GPT-Pilot ([Pythagora.io, 2023](#)), praised for its detailed task delegation via over 20 specialized agents, suffers from reduced interactivity due to an overly granular division of responsibilities, resulting in a more fragmented user experience. These qualitative insights, although not fully captured by quantitative metrics, were evident through the DevAI dataset, providing key areas for improvement in user engagement and operational fluidity in future releases of these frameworks.

## G More DevAI dataset samples

### Task 13: Style Transfer with Perceptual Loss in PyTorch

#### Query

Please create a PyTorch Perceptual Loss project for image style transfer (refer to this paper: [Perceptual Losses for Real-Time Style Transfer](#)). You can build the Perceptual Loss Network using VGG16 in `src/model.py`. The project should combine content and style images, allow smooth adjustment of style intensity by tuning the weights of style loss and content loss, and save the stylized images in `results/figures/`. Additionally, log the processing time to `results/processing_time.txt`, and save the intermediate results of the style transfer process to `results/figures/intermediate_results.png`. For testing, input a famous content image (*Mona Lisa*) from [this link](#) and a famous style image (*The Starry Night*) from [this link](#), and generate a style-transferred image. Save the content, style, and style-transferred images to `data/content.jpg`, `data/style.jpg`, and `results/figures/`, respectively. The project should efficiently handle high-resolution images without excessive processing time.

#### Requirements

##### ■ R0

**Criteria:** A famous content image is inputted for testing, downloaded from [this link](#) and saved to `data/content.jpg`.

**Dependencies** → {}

##### ■ R1

**Criteria:** A famous style image is inputted for testing, downloaded from [this link](#) and saved in `data/style.jpg`.

**Dependencies** → {}

##### ■ R2

**Criteria:** The Perceptual Loss model is implemented in PyTorch and loaded in `src/model.py`.

**Dependencies** → {}

##### ■ R3

**Criteria:** Stylized images are saved to the specified folder `results/figures/`.

**Dependencies** → {R0, R1, R2}

##### ■ R4

**Criteria:** Style intensity is adjusted by tuning the weights of style loss and content loss in `src/model.py`.

**Dependencies** → {R0, R1, R2}

##### ■ R5

**Criteria:** Processing time is recorded and saved as `results/processing_time.txt`.

**Dependencies** → {R0, R1, R2, R3, R4}

##### ■ R6

**Criteria:** Intermediate results of style transfer are saved as `results/figures/intermediate_results.png`.

**Dependencies** → {R0, R1, R2, R3, R4}

#### Preferences (Optional)

##### ■ P0

**Criteria:** The style transfer process should allow for smooth adjustment of style intensity, making the stylized image visually appealing.

##### ■ P1

**Criteria:** The project should handle high-resolution images efficiently without excessive processing time.

Figure 10 An Example Task in DevAI: Task 13.

## Task 19: Time Series Forecasting with Seq2Seq LSTM on Rossmann Store Sales

### Query

Develop a sales forecasting system using a sequence-to-sequence model based on LSTM with the Rossmann Store Sales dataset, downloading it from Kaggle [here](#) and loading it in `src/data_loader.py`. Split the data into training and testing sets and save them in `src/data_loader.py`. Apply a sequence-to-sequence model based on LSTM and save the trained model under the `models/saved_models/` directory. Save the forecast results as `results/figures/forecast_results.png`. Save a comparison plot between the predicted and actual values to `results/figures/comparison_plot.png`. Generate an HTML report that includes the prediction results and comparison plots, with some interactive elements for exploring different forecast horizons, and save it as `results/report.html`. Ensure the model is tuned to capture seasonal trends in the sales data.

### Requirements

#### ■ R0

**Criteria:** The Rossmann Store Sales dataset is used, potentially downloaded from ([this link](#)) and loaded in `src/data_loader.py`.

**Dependencies** → {}

#### ■ R1

**Criteria:** The data is split into training and testing sets and implemented in `src/data_loader.py`.

**Dependencies** → {R0}

#### ■ R2

**Criteria:** A sequence-to-sequence model based on LSTM is used. The trained model should be saved under `models/saved_models/`.

**Dependencies** → {R1}

#### ■ R3

**Criteria:** The forecast results are plotted and saved as `results/figures/forecast_results.png`.

**Dependencies** → {R1, R2}

#### ■ R4

**Criteria:** A comparison plot of predicted vs. actual values is saved as `results/figures/comparison_plot.png`.

**Dependencies** → {R1, R2, R3}

#### ■ R5

**Criteria:** An HTML report containing forecast results and comparison plots is generated and saved as `results/report.html`.

**Dependencies** → {R1, R2, R3, R4}

#### ■ R6

**Criteria:** The HTML report should include interactive elements that allow users to explore different forecast horizons.

**Dependencies** → {R5}

### Preferences (Optional)

#### ■ P0

**Criteria:** The model should be tuned to capture seasonal trends in the sales data for more accurate forecasting.

Figure 11 An Example Task in DevAI: Task 19.

## H Human Evaluation Procedure

We recruited three AI experts from the authors to perform human evaluation on the output of agentic code generation systems. There we present the evaluation details.

*First round* For the first round of evaluations, our three evaluators reported spending 16.5, 19.5, and 22.0 hours, respectively. To capture the bias that a human evaluator will have, the instructions given to our experts were minimal, with them only receiving a scorecard to complete for each agentic system and each task. Results that all evaluators agree on are considered trustworthy. The assumption here is that it is unlikely that all three evaluators make a mistake or have an effective bias in the same judgment. The self-reported post-hoc evaluation criteria are shown in Figure 12.

*Second round* In the second round, the evaluators present and discuss their reasons for disagreeing with judges. In doing so, human errors are likely corrected by their peers. Discussion among evaluators also helps reduce human bias by examining each other’s thought processes thoroughly. Furthermore, the consensed results are considered trustworthy given the assumption that it is unlikely that all three evaluators are convinced by the same mistake or the same cognitive bias. The three evaluators took 9.5 hours together for this second round of evaluation.

## Self-reported Post-hoc Evaluation Criteria after round one

### Evaluator 231a

EXECUTED SUCCESSFULLY: Yes [ ] / No [ ]

1. Must be checked based on the overall completeness of the task, based on looking at the code, the artifacts, and the trajectory.
2. The training has finished, the model snapshot and the metrics breakdown have been saved, and at least one artifact of required analytics has been produced - mark as successful. If some analytic artifacts are missing but not all, mark as successful.
3. No need to run the code.
4. If training was finished but on fake data, mark as successful.

#### Requirements:

Marking a requirement as satisfied must be made for this specific requirement disregarding the dependency list. If a file (code, image, snapshot) is there but is empty or without any meaningful content - mark as No.

#### 1. Code:

The functionality must be in a file with the requested path.

The real data is replaced by simple synthetic - not satisfied.

#### 2. Visualization/Reports:

The contents must be there and make sense even if not perfect from the ML/DS point of view.

#### 3. Snapshots:

If a binary snapshot is not empty, mark as Yes.

### Evaluator 38bb

EXECUTED SUCCESSFULLY: Yes [ ] / No [ ]

An output is marked yes if none of the following is satisfied.

1. The time spent is close to the time limit.
2. The last environmental message includes an error.
3. The last thought indicates that the task is completed.
4. The last step of the trajectory is incomplete.

#### Requirements:

If a required is unsatisfied only because of an unsatisfication of a previously marked unsatisfied requirement, then judge it based on the assumption that a minimum implementation satisfies the previous requirement exists.

#### 1. Code:

Mark yes if the code executes and does the required function. If no entrypoint is given, the evaluator will set an entrypoint. If the code is not executable due to previous unsatisfied requirements, then it is judged based on eye-checking.

#### 2. Visualization/Reports:

Mark yes if the visualization or report exists in a right path and the content aligns the requirement.

#### 3. Snapshots:

Mark yes if the snapshot exists in a right path and is not empty.

### Evaluator cn9o

I evaluated everything based on whether the requirement was satisfied verbatim, using my own professional judgement when there was ambiguity. If there was a data folder and a look at the trajectory indicated it was real, I allowed it to be loaded directly. If there were results and no clear source code making them, I considered the task incomplete. I didn't consider a dummy data path to be correct (GPT-Pilot used a lot of these). To determine if something was executed correctly, I skimmed the end of the logs (e.g., trajectory) for any obvious signs of an error. If there wasn't any, I said it was correctly executed. I ignored prerequisites while evaluating and did not execute anything, instead just eyeballing the code for correctness. I was lenient in what I considered to be sufficient in terms of the more vague requirements (e.g., if the preprocessing had to include scaling and rotation, just those two would be sufficient to consider it done).

**Figure 12** Each evaluator was given a full description of each task and the associated requirements and preferences in markdown format. They were then asked, for each workspace and trajectory generated by each of the agents on each of the task, whether (1) the agent successfully finished its execution cycle and (2) which of the requirements were satisfied. After the evaluation was complete, the evaluators were asked to self-report the nuances of their evaluation.

## I Suggest Constraints

Below is a sample of constraints in JSON format that describes task-specific guidelines:

```
{  
    "generic": "This is a task that requires you to write, execute, and save source  
    code. You have a hard time limit of 30 minutes to produce your programmatic  
    solution to the given task. This time limit includes execution time. The  
    quality of your solution will be judged based on what you left in the  
    working folder by the time 30 minutes expire. Additionally, the hardware you  
    are running on is unknown, and the presence of a GPU is not guaranteed.",  
    "is_training_needed": "Keep the time limit in mind when setting hyperparameters  
    for training.",  
    "is_kaggle_api_needed": "You can use the Kaggle API credentials stored in `  
        kaggle.json` in your current working directory."  
}
```

To address automation and security concerns, we have written code to place the `kaggle.json` file into the current workspace for each baseline during each run. However, dataset users are free to modify the solution to enable the Kaggle API for their specific developer agents.

## J Collected Trajectories

### J.1 Schema

Below is the required JSON format for a trajectory in gray-box settings (where the trajectories can serve as input for LLM-as-a-Judge and Agent-as-a-Judge).

```
{  
    "type": "array",  
    "items": {  
        "type": "object",  
        "properties": {  
            "step": {  
                "type": "integer",  
                "description": "The step number in the trajectory, 0-based."  
            },  
            "user_message": {  
                "type": ["string", "null"],  
                "description": "The message from the external user to the agent. If null,  
                no message was sent."  
            },  
            "agent": {  
                "type": "object",  
                "properties": {  
                    "thought": {  
                        "type": "string",  
                        "description": "The agent's thought at this step."  
                    },  
                    "action": {  
                        "type": ["string", "null"],  
                        "description": "The agent's action sent to the environment. If null,  
                        the agent did not take any action, for example, when the agent has  
                        finished the task."  
                    },  
                    "agent_name": {  
                        "type": "string",  
                        "description": "The name of the agent that made the action."  
                    }  
                },  
                "required": ["thought", "action"],  
                "description": "Everything related to the agent at this step."  
            },  
            "environment": {  
                "type": ["string", "null"],  
                "description": "The environment's (shell, python interpreter) response to  
                the action submitted by the agent. If null, the environment was not  
                involved in this step."  
            },  
            "step_usage": {  
                "type": "object",  
                "properties": {  
                    "input_tokens": {  
                        "type": "integer",  
                        "description": "The number of input tokens passed as LLM context."  
                    },  
                    "output_tokens": {  
                        "type": "integer",  
                        "description": "The number of tokens produced by the LLM."  
                    }  
                },  
                "description": "Information about the tokens used in the step."  
            },  
            "tokens": {  
                "type": "array",  
                "description": "The tokens produced by the LLM, represented as integers.  
                This field is present only if the step_usage object is defined."  
            }  
        },  
        "description": "A single step in a trajectory, containing the user message, the agent's thought and action,  
        the environment's response, and the tokens produced by the LLM."  
    },  
    "description": "A collection of steps, representing a trajectory."  
}
```

```

        "model": {
            "type": "string",
            "description": "The name of the LLM model used."
        },
        "cost": {
            "type": "number",
            "description": "The cost of the LLM inference, in USD."
        },
        "llm_inference_time": {
            "type": "number",
            "description": "The time taken by the LLM to generate the output tokens
                , in seconds."
        },
        "step_execution_time": {
            "type": "number",
            "description": "The time taken to make an entire step including LLM
                inference and environment execution, in seconds."
        }
    },
    "required": [
        "input_tokens",
        "output_tokens",
        "model",
        "cost",
        "llm_inference_time",
        "step_execution_time"
    ]
},
"accumulated_usage": {
    "type": "object",
    "properties": {
        "accumulated_cost": {
            "type": "number",
            "description": "The total cost of the trajectory up to this step, in
                USD."
        },
        "accumulated_time": {
            "type": "number",
            "description": "The total time taken by the agent to complete the
                trajectory up to this step, in seconds."
        }
    },
    "required": [
        "accumulated_cost",
        "accumulated_time"
    ]
},
"required": ["step", "user_message", "agent", "environment", "step_usage", "accumulated_usage"]
}
}

```

## J.2 A Sample of Trajectory

This is sample trajectories (we select 4 steps of it) in OpenHands (Wang et al., 2024d). All steps are recorded to enable replay of the trajectory.

```
{  
    "step": 0,  
    "user_message": "Help me develop an object detection system using the YOLOv3 model and the COCO dataset. Download the dataset and preprocess the images by resizing and normalization in `src/data_loader.py`. Implement the YOLOv3 model and use Non-Maximum Suppression (NMS) to refine the results in `src/model.py`. Save the detected objects to `results/figures/`, and create an interactive Streamlit web page in `src/app.py` to display the detection results. Finally, evaluate the model's performance, including metrics such as mAP and inference time, and save the evaluation results to `results/metrics/model_performance.txt`. The system should properly manage the launch and termination of the Streamlit application to prevent unnecessary resource usage.\n\nThis is a task that requires you to write, execute, and save source code. You have a hard time limit of 30 minutes to produce your programmatic solution to the given task. This time limit includes execution time. The quality of your solution will be judged based on what you left in the working folder by the time 30 minutes expire. Additionally, the hardware you are running on is unknown, and the presence of a GPU is not guaranteed.\n\nKeep the time limit in mind when setting hyperparameters for training.\n",  
    "agent": {  
        "thought": "Let's break down the task into manageable steps and start implementing them one by one.\n\n### Steps:  
1. **Set up the directory structure:** - Create necessary directories: `src/` and `results/figures/`.  
2. **Download and preprocess the COCO dataset:** - Implement `src/data_loader.py` to download, resize, and normalize images.  
3. **Implement the YOLOv3 model:** - Implement `src/model.py` to define the YOLOv3 model and apply Non-Maximum Suppression (NMS).  
4. **Create an interactive Streamlit web page:** - Implement `src/app.py` to display detection results.  
5. **Evaluate the model's performance:** - Save evaluation metrics to `results/metrics/model_performance.txt`.  
Let's start by setting up the directory structure and creating the necessary files.\n\n### Step 1: Set up the directory structure",  
        "action": "Running command: mkdir -p src results/figures results/metrics\ntouch src/data_loader.py src/model.py src/app.py"  
    },  
    "environment": "\r\n[Python Interpreter: /openhands/poetry/openhands-504_aCHf-py3.11/bin/python]\nopenhands@b34322eae8da:/workspace $ touch src/data_loader.py src/model.py src/app.py\r\n[Python Interpreter: /openhands/poetry/openhands-504_aCHf-py3.11/bin/python]\nopenhands@b34322eae8da:/workspace $ ",  
    "step_usage": {  
        "input_tokens": 4331,  
        "output_tokens": 220,  
        "llm_inference_time": 4.445789337158203,  
        "model": "gpt-4o-2024-05-13",  
        "cost": 0.024955,  
        "step_execution_time": 16.24921226501465  
    },  
    "accumulated_usage": {  
        "accumulated_cost": 0.024955,  
        "accumulated_time": 16.24922204017639  
    }  
}
```

```

        }
    },
    {
        "step": 1,
        "user_message": null,
        "agent": {
            "thought": "Great! The directory structure and necessary files are created. Let's proceed with implementing the data loader to download and preprocess the COCO dataset.\n\n## Step 2: Implement `src/data_loader.py`\nWe will:\n1. Download the COCO dataset.\n2. Resize and normalize the images.\nLet's start by writing the code for `src/data_loader.py`.",
            "action": "Running Python code interactively: create_file('src/data_loader.py')"
        },
        "environment": "FileExistsError"
    },
    "environment": "FileExistsError"
},
"environment": "[File: /workspace/src/data_loader.py (1 lines total)]\nthis is the beginning of the file\n1|\n(this is the end of the file)\n[Jupyter current working directory: /workspace]\n[Jupyter Python interpreter: /openhands/poetry/openhands-504_aCHf-py3.11/bin/python]",
"step_usage": {
    "input_tokens": 4675,
    "output_tokens": 84,
    "llm_inference_time": 2.136143207550049,
    "model": "gpt-4o-2024-05-13",
    "cost": 0.02463500000000004,
    "step_execution_time": 3.345384359359741
},
"accumulated_usage": {
    "accumulated_cost": 0.04959000000000001,
    "accumulated_time": 19.594608068466187
}
},
{
    "step": 2,
    "user_message": null,
    "agent": {
        "thought": "The file `src/data_loader.py` already exists. Let's open it and start writing the code to download and preprocess the COCO dataset.\n\n## Step 2: Implement `src/data_loader.py`",
        "action": "Running Python code interactively: open_file('src/data_loader.py')"
    },
    "environment": "[File: /workspace/src/data_loader.py (1 lines total)]\nthis is the beginning of the file\n1|\n(this is the end of the file)\n[Jupyter current working directory: /workspace]\n[Jupyter Python interpreter: /openhands/poetry/openhands-504_aCHf-py3.11/bin/python]",
    "step_usage": {
        "input_tokens": 4982,
        "output_tokens": 53,
        "llm_inference_time": 2.209756851196289,
        "model": "gpt-4o-2024-05-13",
        "cost": 0.02570500000000002,
        "step_execution_time": 3.345384359359741
    }
}
}

```

```

        "step_execution_time": 2.318861961364746
    },
    "accumulated_usage": {
        "accumulated_cost": 0.07529500000000001,
        "accumulated_time": 21.913471937179565
    }
},
...
{
    "step": 20,
    "user_message": null,
    "agent": {
        "thought": "It seems the indentation issue persists. Let's correct the indentation manually.\n\n### Corrected Code for `src/evaluate.py`\n```\npython\nimport time\nimport numpy as np\nfrom model import\nyolo_v3_model, non_max_suppression\nfrom data_loader import\npreprocess_image, download_coco_dataset, extract_coco_dataset\n\nimport os\n\n\ndef evaluate_model():\n    model = yolo_v3_model()\n\n    download_coco_dataset()\n    extract_coco_dataset()\n\n    image_dir = \"data/val2017\"\n    image_files = [os.path.join(\n        image_dir, f) for f in os.listdir(image_dir) if f.endswith('.jpg')]\n\n    total_inference_time = 0\n    num_images = len(image_files)\n\n    all_boxes = []\n    all_scores = []\n    all_classes = []\n\n    for image_file in image_files:\n        image =\n            preprocess_image(image_file)\n        image = np.expand_dims(image,\n            axis=0)\n\n        start_time = time.time()\n        predictions =\n            model.predict(image)\n        end_time = time.time()\n\n        inference_time = end_time - start_time\n\n        total_inference_time += inference_time\n\n        boxes, scores,\n        classes = non_max_suppression(predictions[0])\n\n        all_boxes.append(boxes)\n        all_scores.append(scores)\n\n        all_classes.append(classes)\n\n        mean_inference_time =\n            total_inference_time / num_images\n\n        # Placeholder for mAP\n        calculation\n        mAP = 0.0\n        # This should be replaced with actual\n        mAP calculation\n\n        with open(\"results/metrics/\n            model_performance.txt\", \"w\") as f:\n            f.write(f\"Mean\n            Inference Time: {mean_inference_time:.4f} seconds\\n\")\n\n        f.write(f\"mAP: {mAP:.4f}\\n\")\n\n        if __name__ == \"__main__\":\n            evaluate_model()\n```\nLet's update the file with the corrected code.",
        "action": "Running Python code interactively: EDITED_CODE = \"\"\nimport time\nimport numpy as np\nfrom model import yolo_v3_model,\nnon_max_suppression\nfrom data_loader import preprocess_image,\ndownload_coco_dataset, extract_coco_dataset\n\nimport os\n\n\ndef evaluate_model():\n    model = yolo_v3_model()\n\n    download_coco_dataset()\n    extract_coco_dataset()\n\n    image_dir = \"data/val2017\"\n    image_files = [os.path.join(\n        image_dir, f) for f in os.listdir(image_dir) if f.endswith('.jpg')]\n\n    total_inference_time = 0\n    num_images = len(image_files)\n\n    all_boxes = []\n    all_scores = []\n    all_classes = []\n\n    for image_file in image_files:\n        image =\n            preprocess_image(image_file)\n        image = np.expand_dims(image,\n            axis=0)\n\n        start_time = time.time()\n        predictions =\n            model.predict(image)\n        end_time = time.time()\n\n        inference_time = end_time - start_time\n\n        total_inference_time += inference_time\n\n        boxes, scores,\n        classes = non_max_suppression(predictions[0])\n\n        all_boxes.append(boxes)\n        all_scores.append(scores)\n\n        all_classes.append(classes)\n\n        mean_inference_time =\n            total_inference_time / num_images\n\n        # Placeholder for mAP\n        calculation\n        mAP = 0.0\n        # This should be replaced with actual\n        mAP calculation\n\n        with open(\"results/metrics/\n            model_performance.txt\", \"w\") as f:\n            f.write(f\"Mean\n            Inference Time: {mean_inference_time:.4f} seconds\\n\")\n\n        f.write(f\"mAP: {mAP:.4f}\\n\")\n\n        if __name__ == \"__main__\":\n            evaluate_model()"
    }
}

```

```

        calculation\n      mAP = 0.0 # This should be replaced with actual
mAP calculation\n\n      with open(\"results/metrics/
model_performance.txt\", \"w\") as f:\n          f.write(f\"Mean
Inference Time: {mean_inference_time:.4f} seconds\\\\\\n\")\n
f.write(f\"mAP: {mAP:.4f}\\\\\\n\")\nif __name__ == \"__main__\":\n
    evaluate_model()\n\"\"\nedit_file_by_replace('src/evaluate.
py', 'with open(\"results/metrics/model_performance.txt\", \"w\")
as f:', '      with open(\"results/metrics/model_performance.txt\", \
\"w\") as f:\")"
},
"environment": null,
"step_usage": {
    "input_tokens": 20132,
    "output_tokens": 805,
    "llm_inference_time": 15.1565523147583,
    "model": "gpt-4o-2024-05-13",
    "cost": 0.11273500000000002,
    "step_execution_time": 15.196227073669434
},
"accumulated_usage": {
    "accumulated_cost": 1.2919750000000003,
    "accumulated_time": 176.99772310256958
}
}
}

```

## K Ablations of Agent-as-a-Judge

### K.1 Component Abalations

**Table 5 Component Ablation Studies for Agent-as-a-Judge.** We analyze the impact of sequentially adding various components (`ask`, `graph`, `read`, `locate`, `retrieve`, `planning`, `memory`, and `search`) on the performance of Agent-as-a-Judge for judging OpenHands.

Metric	+ ask	+ graph	+ read	+ locate	+ search	+ retrieve	+ planning	+ memory
<b>Agent-as-a-Judge Performance</b>								
<b>Alignment Rate</b>	65.03%	75.95%	82.24%	90.44%	86.06%	90.16%	88.52%	87.97%

**Analysis** We designed 8 modular components for the Agent-as-a-Judge system. In the [Table 5](#), components are added progressively from left to right. If the addition of a component led to a significant performance drop, we removed it from further iterations. Our experiments showed that adding the components `ask`, `graph`, `read`, and `locate` resulted in significant performance gains. However, when the `search` component was introduced, there was a noticeable decline in performance.

We hypothesize that the performance drop from `search` is due to its role in retrieving relevant code snippets (top-3) using BM25. The retrieval accuracy of BM25 ([Robertson et al., 2009](#)) might not have been high enough, potentially introducing noise. Moreover, as noted in [Table 1](#), the DevAI tasks in our experiments did not generate a large amount of code. In fact, even when all code was fed into an LLM, the total content typically stayed within the maximum context length. Therefore, in simpler workspaces, `search` was less critical. However, we believe this component will become more important as the complexity of the workspace increases, making it more valuable in larger and more complex environments.

We also observed that the introduction of the `planning` mechanism did not bring a noticeable improvement in performance. This may be related to the nature of the Judge - it needs clean factual information. When `planning` is unstable, the evidence collected from different actions can become inconsistent, leading to a decline in performance. Finally, we experimented with a `memory` mechanism. Initially, we hypothesized that since DevAI tasks often involve interconnected requirements, `memory` could help track whether requirements were met. However, in practice, we saw no improvement. We suspect that the interconnected nature of the requirements may have caused biases: specifically, once a prior requirement was fulfilled, it might have overly influenced positive judgments on subsequent requirements, even if they were not fully met.

### K.2 Search Algorithms in Search Module

We initially hypothesized that the performance drop was due to the low precision of the `search` component, particularly with BM2.5. To explore this, we replaced BM2.5 with Sentence-BERT ([Reimers, 2019](#)) as a more advanced alternative and tested Fuzzy Search ([Levenshtein, 1966](#)) as a less precise option. However, neither improved the performance of the Agent-as-a-Judge.

These results suggest that the performance issue is not due to BM2.5's poor search accuracy. Instead, the workspaces generated in our DevAI tasks are too simple for the `search` component to have a significant impact. In simpler workspaces, direct retrieval and evaluation are sufficient. Even though Sentence-BERT performed better than the other methods, its alignment rate (87.70%) still falls short of the configuration without the `search` component (90.44%). As workspace complexity increases, the `search` component may become more valuable.

**Table 6** Comparisons on `Search` module with different engines.

Search Method	Alignment Rate
BM2.5	86.06%
Sentence-BERT	87.70%
Fuzzy Search	85.52%
<i>without</i> Search Module	90.44%

### K.3 Search Algorithms in Retrieve Module

In our experiments, we found that accurately locating relevant information within a trajectory is a challenging task. Although the addition of the `retrieve` component (gray-box) did not lead to a significant improvement in performance in this specific case, its impact has been notable in other settings, such as in GPT-Pilot. As shown in [Table 3](#), the integration of `retrieve` in GPT-Pilot brought substantial gains.

We conducted an ablation study on GPT-Pilot to optimize the retrieval of useful information at each step. Our experiments revealed that in large trajectories, truncating the final sections of the file often results in losing critical information, as the latter part of the trajectory typically contains dense information about the final development state. Truncating the beginning of the trajectory proved to be the most effective in improving the retrieval efficiency.

For individual steps, truncating the middle section worked best. This is because error messages usually appear early in the output, while the corresponding file paths and specific error locations are found towards the end. By focusing on these retrieval strategies, we can significantly enhance the performance of the `retrieve` component, particularly in complex scenarios like GPT-Pilot.

**Table 7** Ablations on `retrieve`.

Method	Alignment Rate
Without <code>retrieve</code>	83.88%
With <code>retrieve</code> (gray-box)	86.61%
Trajectory Truncate (head)	86.61%
Trajectory Truncate (middle)	85.52%
Trajectory Truncate (tail)	82.51%
Step Truncate (head)	86.34%
Step Truncate (middle)	86.61%
Step Truncate (tail)	83.88%

## L Prompt Demos of Agent-as-a-Judge

Here, we present some prompts used by the Agent-as-a-Judge system. Each of these prompt demos plays a crucial role in guiding the agent's behavior.

### L.1 System Prompt for Agent-as-a-Judge

```
def get_system_prompt(language="English"):

    if language == "English":
        """
        You are an advanced AI system serving as an impartial judge for intelligent
        code generation outputs. Your primary role is to rigorously evaluate
        whether the agent's outputs satisfy the specified requirements by
        thoroughly analyzing the provided code, data, and other relevant
        materials.

        You will systematically assess aspects such as datasets, model
        implementations, training procedures, and any task-specific criteria
        outlined in the requirements. Your evaluations must be objective,
        detailed, and based solely on the evidence provided.

        For each requirement, deliver one of the following judgments:

        1. <SATISFIED>: Use this if the agent's output fully meets the requirement.
           Provide a brief and precise explanation demonstrating how the specific
           criteria are fulfilled.

        2. <UNSATISFIED>: Use this if the agent's output does not meet the
           requirement. Provide a concise explanation indicating the deficiencies
           or omissions.

        Your assessment should reference specific elements such as code snippets,
        data samples, or output results where appropriate. Ensure that your
        justifications are clear, precise, and directly related to the criteria

        .

        Respond with either <SATISFIED> or <UNSATISFIED>, followed by your concise
        justification.
        """
    else:
        raise NotImplementedError(f"The language '{language}' is not supported.")
```

## L.2 System Prompt for Locate Module

```
def get_locate_system_prompt(language="English"):  
    if language == "English":  
        return """  
        You are an advanced AI system specializing in understanding project  
        structures and determining file locations based on provided criteria.  
        Your task is to locate specific files in the workspace based on the user's  
        criteria and workspace information.ution problems with the files  
        mentioned in the criteria.  
        """  
  
    else:  
        raise NotImplementedError(f"The language '{language}' is not supported.")
```

## L.3 System Prompt for Retrieve Module

```
def get_retrieve_system_prompt(language="English"):  
  
    if language == "English":  
        return """  
        You are an advanced AI system specializing in retrieving environmental  
        feedback from project execution trajectories. Your task is to analyze  
        the provided trajectory data and extract information about the most  
        relevant files mentioned in the given criteria.  
  
        Focus on the following:  
  
        1. Identify the **most recent steps** where the files directly related to  
           the criteria were involved in execution, loading, or saving operations.  
        2. Provide environmental feedback for these files, such as any errors,  
           warnings, or issues encountered during their execution or processing.  
        3. Highlight whether any problems occurred that might affect the  
           functionality or success of these files in the project.  
  
        Your output should be structured as follows:  
  
        - **<RELEVANT STEPS>**: List the specific steps involving the relevant  
          files, including any environmental feedback such as error messages,  
          execution results, or other issues encountered. Each step should  
          concisely present the key information needed to assess the files'  
          execution status.  
  
        Avoid including details about file contents or existence, as this  
        information is already available. Focus solely on the environmental  
        feedback related to the execution of the most relevant files.  
  
        Your goal is to provide clear and concise information that helps determine  
        if there were any execution problems with the files mentioned in the  
        criteria.  
        """  
  
    else:  
        raise NotImplementedError(f"The language '{language}' is not supported.")
```

## L.4 Prompt for Ask Module (for requirement check)

```
def get_ask_prompt(criteria: str, evidence: str) -> str:  
  
    return f"""  
Provided below is relevant information about the project:  
{evidence}  
  
Kindly perform an evaluation of the following criteria:  
{criteria}  
  
As per the guidelines, respond with either <SATISFIED> or <UNSATISFIED>,  
followed by a concise justification that references specific elements from  
the project information, such as code snippets, data samples, or output  
results.  
"""
```

## L.5 Prompt for Locate Module

```
def get_locate_prompt(criteria: str, evidence: str) -> str:  
  
    return f"""  
Provided below is the structure of the workspace:  
{workspace_info}  
  
This is the criteria related to the task:  
{criteria}  
  
Follow the format in the example below and return only the file paths that  
match the criteria:  
  
Example:  
  
Suppose the criteria is:  
'The database functionality is implemented in `src/db.py`, and the logging  
system is defined in `src/logging.py`.'  
  
And the workspace information is:  
/project  
|-- src  
|   |-- db.py  
|   |-- logging.py  
|   |-- utils.py  
|-- tests  
   |-- test_db.py  
   |-- test_logging.py  
  
Based on the criteria, the following paths (no more than 5) should be returned,  
each wrapped in dollar signs (`$`):  
$/project/src/db.py$  
$/project/src/logging.py$  
"""
```

## M Judge Evidences Collected from Agent-as-a-Judge

The Agent-as-a-Judge system dynamically collects evidence throughout the development process of the code-generating agentic system to evaluate whether each requirement is satisfied. Using this auxiliary information, the judge agent decides to focus on relevant data such as code, file structures, and trajectories. This gathered evidence supports the agent's final judgment, ensuring an informed and accurate assessment of the project's compliance with its requirements. A sample of the collected evidence is shown in the following [Appendix M](#).

```

1  >>> [Reference] The Following Is the Original User Query:
2
3  Please help me build a system for sentiment analysis on tweets using the Sentiment140 dataset available from Hugging Face. Load the Sentiment140 dataset and,
4  when loading the data, clean it by removing stop words, punctuation, and special characters, all in 'src/data_loader.py'. Use Word2Vec or Glove for text
5  vectorization. This should occur in 'src/data_loader.py'. Next, implement and train an SVM classifier in 'src/model.py'. Finally, write a report of the accuracy
6  of the classifier to 'results/metrics/accuracy_score.txt'. Ideally, the report should be easily interpretable.
7
8  >>> [Key Evidence] The Following Is the Project and Workspace Structure Information:
9
10 | Project Tree
11 |
12 | Workspace Path: OpenHands/workspaces/06_Sentiment_Analysis_SVM_Sentiment140_ML
13 |
14 | Total Nodes: 5
15 |
16 | Project Structure
17 |
18 |   results
19 |   results/metrics
20 |   results/model
21 |   src
22 |     data_loader.py
23 |     model.py
24
25
26  >>> [Reference] The Following Is Relevant Search Evidence:
27
28 | File: OpenHands/workspaces/06_Sentiment_Analysis_SVM_Sentiment140_ML/src/data_loader.py
29 | Lines: [26, 28]
30 | Identifier: ref
31 | Category: function
32 |
33 | 26     dataset = load_and_preprocess_data()
34 |
35 | 27
36
37  >>> [Code Snippet]
38
39 | File: OpenHands/workspaces/06_Sentiment_Analysis_SVM_Sentiment140_ML/src/model.py
40 | Lines: [18, 19]
41 | Identifier: ref
42 | Category: function
43 |
44 | 18     dataset = load_and_preprocess_data()
45 |
46 | 19
47
48  >>> [Code Snippet]
49
50 | File: OpenHands/workspaces/06_Sentiment_Analysis_SVM_Sentiment140_ML/src/data_loader.py
51 | Lines: [21, 28]
52 | Identifier: def
53 | Category: function
54 |
55 | 21 def load_and_preprocess_data():
56 | 22     # Load the Sentiment140 dataset
57 | 23     dataset = pd.read_csv('sentiment140', split='train[:1%]', trust_remote_code=True)
58 | 24     # Clean the text data
59 | 25     dataset = dataset.map(lambda x: {'text': clean_text(x['text'])})
60 | 26     return dataset
61
62
63
64
65  >>> [Key Evidence] The Following Is the Contents of the Mentioned Files:
66
67 Content of OpenHands/workspaces/06_Sentiment_Analysis_SVM_Sentiment140_ML/src/data_loader.py:
68
69 import re
70 import nltk
71 from datasets import load_dataset
72 from nltk.corpus import stopwords
73 from nltk.tokenize import word_tokenize
74
75 nltk.download('stopwords')
76 nltk.download('punkt')
77
78 def clean_text(text):
79     # Remove special characters and punctuation
80     text = re.sub(r'[^a-zA-Z\s]', '', text)
81     # Tokenize the text
82     tokens = word_tokenize(text)
83     # Remove stop words
84     tokens = [word for word in tokens if word.lower() not in stopwords.words('english')]
85     return ' '.join(tokens)
86
87 def load_and_preprocess_data():
88     # Load the Sentiment140 dataset
89     dataset = load_dataset('sentiment140', split='train[:1%]', trust_remote_code=True)
90     # Clean the text data
91     dataset = dataset.map(lambda x: {'text': clean_text(x['text'])})
92     return dataset
93
94
95 if __name__ == "__main__":
96     dataset = load_and_preprocess_data()
97     print(dataset[0])
98
99
100 >>> [Reference] The Following Is Development Trajectory Evidence:
101
102 >>> Relevant Steps in Trajectory
103
104 | The following environment feedback is provided for reference only and does
105 | not serve as decisive evidence. For final judgement, please rely primarily
106 | on local file contents.
107 |
108 | =>Step 21=> The 'src/data_loader.py' script was executed to preprocess
109 | the Sentiment140 dataset. The environment feedback indicated successful
110 | execution of the script, with the dataset being loaded and preprocessed.
111 | The output included the cleaned text data, which removed stop words,
112 | stopwords and punt, which are essential for text preprocessing tasks like
113 | removing stop words and tokenization.
114 |
115 | =>Step 23=> The 'src/data_loader.py' script was run again to verify the
116 | preprocessing steps. The environment feedback confirmed that the
117 | preprocessing was completed successfully, and the first example from the
118 | dataset was printed, indicating that the dataset was loaded and processed
119 | correctly.
120 |
121 | Overall, the preprocessing in 'src/data_loader.py' was executed without
122 | errors, and the necessary steps for cleaning the dataset, such as removing
123 | stop words, punctuation, and special characters, were implemented and
124 | verified.
125 |
126
127
128 >>> [Reference] The Following Is Historical Judgments:
129
130 | Requirement 1: The "Sentiment140" dataset, available from "Hugging Face," is obtained in 'src/data_loader.py'.
131 | Satisfied: True
132

```