TYPE Original Research
PUBLISHED 15 May 2025
DOI 10.3389/frai.2025.1431003

OPEN ACCESS

EDITED BY
Mayank Singh,
Indian Institute of Technology Gandhinagar, India

REVIEWED BY
Simona Frenda,
University of Turin, Italy
Xiaohao Wen,
Guangxi Normal University, China

*CORRESPONDENCE
Raymond Lee
✉ raymondshtlee@uic.edu.cn

RECEIVED 11 May 2024
ACCEPTED 30 April 2025
PUBLISHED 15 May 2025

CITATION
Yu J, Wu Y, Zhan Y, Guo W, Xu Z and Lee R (2025) Co-Learning: code learning for multi-agent reinforcement collaborative framework with conversational natural language interfaces.
Front. Artif. Intell. 8:1431003.
doi: 10.3389/frai.2025.1431003

Co-Learning: code learning for multi-agent reinforcement collaborative framework with conversational natural language interfaces

Jiapeng Yu, Yuqian Wu, Yajing Zhan, Wenhao Guo, Zhou Xu and Raymond Lee*

Guangdong Provincial Key Laboratory of Interdisciplinary Research and Application for Data Science, Faculty of Science and Technology, Beijing Normal University-Hong Kong Baptist University United International College, Zhuhai, China

Online question-and-answer (Q&A) systems based on the Large Language Model (LLM) have progressively diverged from recreational to professional use. However, beginners in programming often struggle to correct code errors independently, limiting their learning efficiency. This paper proposed a Multi-Agent framework with environmentally reinforcement learning (E-RL) for code correction called Code Learning (Co-Learning) community, assisting beginners to correct code errors independently. It evaluates the performance of multiple LLMs from an original dataset with 702 error codes, uses it as a reward or punishment criterion for E-RL; Analyzes input error codes by the current agent; selects the appropriate LLM-based agent to achieve optimal error correction accuracy and reduce correction time. Experiment results showed that 3% improvement in Precision score and 15% improvement in time cost as compared with no E-RL method respectively. The results indicate that integrating E-RL with a multi-agent selection strategy can effectively enhance both the accuracy and efficiency of LLM-based code correction systems, making them more practical for educational and professional programming support scenarios.

KEYWORDS

multi-agent, large language model, reinforcement learning, prompting, education

1 Introduction

Large Language Model (LLM)-based conversational question-answering systems, such as Chat Generative Pre-trained Transformer (ChatGPT), have become prominent deep learning networks capable of addressing a wide range of tasks—everyday inquiries to professional task solutions (Nijkamp et al., 2022; Barrault et al., 2023). They can demonstrate reasoning and planning strengths to match an autonomous agent's definition to perceive its surroundings, make decisions, operate, and even build multi-agents to solve complex problems (Xi et al., 2023; Zhou et al., 2023). The majority of multi-agent frameworks can usually complete stationary streaming tasks using fixed prompts but are unable to select the optimal agent according to specific task content (Wang et al., 2024).

Program coding involves time-consuming professional skills due to the specific task's requirement. Beginners often strive for code understanding but may cede programming due to the lack of guidance to resolve unforeseen errors. This study proposes a *Code-Learning* community based on an LLM multi-agent framework on code correction and annotation for efficient learning in communication with users. It uses reinforcement learning to decide which agent is required for the next step based on the input problem or the output generated by the current agent, in contrast to previous multi-agent code generation, error-correction networks based on a defined single stream (Chen et al., 2023). There are five agents responsible for different tasks: (1) Main agent supervises and exchanges information with users, (2) Correction agent revises programming, (3) Interpretation agent explains the programming logic to subsequent agents to locate incorrect codes, (4) Test agent generates correct codes, and (5) Annotation agent adds comments to the revised code for user's understanding. These five agents communicate through conversation interfaces. The multi-agent generated by the main one is a copy with E-RL to self-improve and provide feedback to both counterparts and human users. Co-Learning uses ERNIE (Sun et al., 2021), SparkDesk (iFLYTEK, 2023), and LLaMa (Touvron et al., 2023) as base models for different agents. Code error correction with E-RL performance is evaluated by passing probability tests, single loop computation time, and the number of loops required. The annotation results are evaluated by an expert reviewer through the location, accuracy, and comprehensibility of annotations.

The aims of this study are to:

1 build a multi-agent framework based on multi-LLMs for code error correction.
2 use original error code datasets to evaluate the performance of multiple LLMs.
3 explore the possibility of reinforcement learning for a large language model-based multi-agent operating environment.
4 compare benchmark frameworks to indicate significant accuracy and operating speed improvements.

2 Related studies

2.1 Prompting with feedback

Recent research on large language models has shown that effective use of prompt words can reduce adverse output (Ganguli et al., 2023) and induce LLM to generate crucial assessments (Fang et al., 2024). Prompt engineering is a specialized study with remarkable benefits for reasoning-type tasks (Madaan et al., 2024). Reflexion (Shinn et al., 2023) pointed out that using linguistic feedback can reinforce LLM instead of weights to store the feedback text in memory and induce the large language model to make better decisions, allowing the language agent to learn by mistakes efficiently. Dialog-Enabled Resolving Agents (DEAR) (Nair et al., 2023) can improve LLM judgment in clinical medicine by simulating two agents converse with each other so that the researcher agent can process information and extract key points of the problem and the decision-maker agent integrates them from the researcher agent to judge the final output accordingly. Self-debugging (Chen et al., 2023) interprets its self-generated code, assisting LLM in identifying code errors without explicitly pointing out the errors and modifications by mimicking a rubber duck test performed by human programmers without extra instructions.

Co-Learning also includes the thinking of prompting with feedback. When the generated code fails to pass the test, Co-Learning interprets and modifies its self-generated code according to the memorized linguistic feedback. At the same time, reinforcement learning will automatically select the optimal agent for the next action based on the feedback from the current agent.

2.2 Multi-agent framework

Multi-agent frameworks emerged at the end of the 20th century (Bradshaw, 1997) when software engineers used Java to write multi-agents for computers to perform by splitting into small, separate tasks that allow agents to focus and cooperate with each other. At the beginning of the 21st century, Java Agent Development Framework (JADE) (Bellifemine et al., 2001; Lee, 2005) standardized the multi-agent forms based on Java, which was used in finance, trading, and journalism (Lee and Liu, 2001).

Python Agent DEvelopment (PADE), a multi-agent framework based on Python (Melo et al., 2019), can be seen as a Python implementation of JADE, which re-implements JADE's core functionalities using Python, making it more suitable for projects that rely on Python-based environments.

LLM, led by GPT (OpenAI, 2023), showed enormous potential, making it possible to employ LLM instead of programs to create agents (Li et al., 2023). LLM's cognitive abilities in single agents have provided a multi-agent foundation (Sumers et al., 2023). Many experiments have shown that complex, dynamic tasks can be completed by multiple large language model agents equipped with strategies and communications (Zhang et al., 2023).

Hence, Co-Learning uses a PADE framework to create agents with functions, whereas multiple LLMs are the core component of a multi-agent framework and information transfer between individual agents to achieve a dynamic workflow.

2.3 Reinforcement learning

Reinforcement learning (RL) is a machine learning paradigm where agents interact with environments to maximize cumulative rewards (Kaelbling et al., 1996). While traditional RL approaches, such as deep deterministic policy gradients (Lillicrap et al., 2015), have shown success in domains ranging from game-playing (Silver et al., 2016) to autonomous driving (Kiran et al., 2021), their adaptation to language models introduces unique challenges. Current RL for LLMs predominantly relies on reinforcement learning from human feedback (RLHF) that requires labor-intensive preference labeling (Schick and Schütze, 2020) or memory-augmented parameter updates (Nair et al., 2023; Chen et al., 2023), incurring substantial human and computational costs. These methods still necessitate parameter fine-tuning and Graphics Processing Unit (GPU)-intensive computations, creating hardware bottlenecks for practical deployment, especially when integrating closed-source models such as Spark API, where fine-tuning is prohibitively expensive. Our framework advances this direction through three key innovations: (1) Full automation of the evaluation-reward cycle using predefined test cases and performance metrics, eliminating human-in-the-loop requirements; (2) Parameter-free adaptation through dynamic agent selection rather than weight updates; and (3) Hardware-agnostic operation via API orchestration, enabling CPU-only execution. This differs fundamentally from prompt-based RL approaches that still require human-curated reward signals and establishes a new paradigm for lightweight RL integration with LLMs.

3 Methodology

3.1 Proposed multi-agent code correction framework (Co-Learning)

A Co-Learning multi-agent code correction framework is illustrated in Figure 1. It has a coexistent framework that relies on PADE (Melo et al., 2019), and the entire workflow runs in the environment created by the main agent. To begin, a correction agent uses a default large language model to make an initial modification for the input error code, returns the generated sentence, and transmits it to the test agent. The test agent performs tests based on the test samples from the dataset mentioned in Section 4.1, which includes the error code and the test cases that should be passed for the code after correction. Specifically, the test agent dynamically executes the Python code generated by the correction agent and evaluates its correctness and score by running it through the basic and challenging test cases provided in the dataset. If the code passes all tests, meaning the generation is correct, the test agent will send the code to the annotation agent for annotation and output it as the correct code. If the code is unable to pass any test, the generated code will be passed to the interpretation agent, and the interpretation is stored in memory as an environmental reinforcement learning prompt. Then, an error code will transfer to the correction agent selected by reinforcement learning to re-generate a code based on the memorized code and interpretation. A loop will be formed by passing the generated result back to the test agent. Error codes entered by outsiders during actual use are not included in the test cases in the test dataset; three forms of tests used by the test agent will be provided: test samples entered by the user, test samples generated by LLM based on user-typing-requirement, and the code correctness determined directly by LLM.

For different agents, the main agent stores all hyper-parameters and historical information, using E-RL based on other agent feedback to update the state of the environment. The test agent creates namespaces to declare the generated code, uses test cases to check the generated code, and returns test results and error messages. The rest of the agents clarify their tasks according to the prompt words, combine them with historical information to generate input streams for the LLM, and return results to the main agent for storage. Co-Learning involves agents cooperating with each other, mimicking human rubber duck testing while using unit test feedback, and selecting the most appropriate large language model in a real-time manner based on E-RL to enhance the performance of code error correction.

3.2 Python agent development (PADE)

Python Agent DEvelopment (PADE) is a simple Python-based approach to create agents that can be accessed by different devices (Melo et al., 2019). This enables the development and creation of communication networks among different agents in accordance with the Foundation for Intelligent Physical Agents (FIPA) standards. PADE is an architecture based on Twisted to develop a multi-agent application using its library resources (Library) and perform a Running Environment of a distributed system. PADE controls the platform by creating an agent (Agent Management System) responsible for platform operations, realize for internal platform functions, and migrate agents out of the platform to other platforms.

A PADE architecture is depicted in Figure 2. It consists of seven modules with the following functions:

a Core: All agents will inherit this base agent kernel framework when created;
b Behaviors: A behavioral template implemented by the agent can be inherited by the user to define a variety of personalized behaviors according to FIPA standards;
c Agent Communication Language (ACL): A language model set up (?) for information interaction between agents according to FIPA standards;
d WEB: A Web server with a graphical interface for interaction with registered sessions, agents, and message databases;
e CLI: Functions interaction with the PADE platform;
f Miscellaneous (Misc): General functions such as looping a standard form of agent initialization message on the screen;
g Tests: Module testing.

The core of the PADE framework is agent execution. Figure 3 illustrates an agent execution UML structure, which indicates the class agent and its interaction with other class agents.

All agents use the PADE framework (e.g., AgentFactory, AgentProtocol in Figure 3), inherit the agent template, and conform to the Twisted protocol. They are identified by their own Agent IDentifier (AID). An agent can be seen as a connected node in the server platform network that can initiate message exchanges or respond to requests from other network nodes via their AIDs.

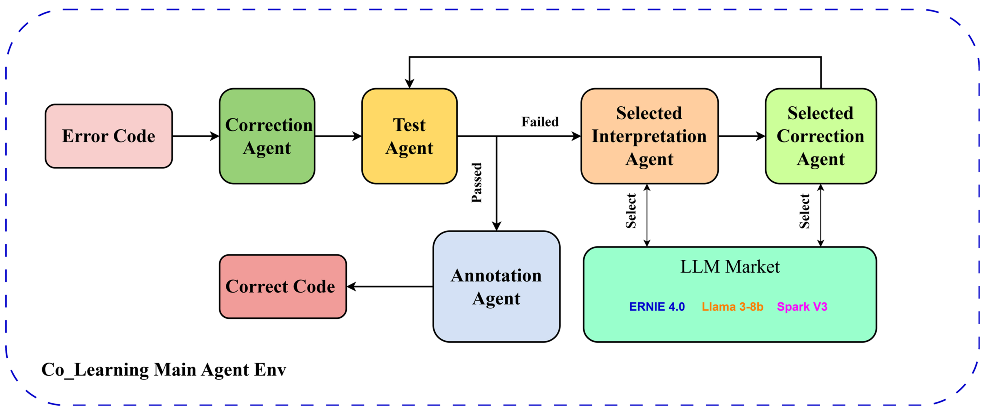


FIGURE 1
Framework of multi-agent code correction (Co-Learning).

An Agent Management System (AMS) in PADE implements key functions such as control and supervision through a table containing AIDs of all agents according to the FIPA 00023 standards. As an agent, it is the first one to activate, and the rest of the agents are required to register and activate. Each agent in the network is required to send a message to the AMS agent so that its AID can be saved as a text string by the AMS. Then, each agent in the platform can access to a table in AMS that stores the names and addresses of all agents to identify other agents. The AMS is updated and distributed whenever an agent enters or exits the network.

Figure 4 illustrates an example where the AMS is the first agent to be launched. It performs registration for agents 1, 2, and 3 as they enter, and informs the existing agents when a new agent joins the network. In this case, when agent 3 fails to register for the first time, its address cannot communicate with other agents. It can only send to the other agents after the second attempt is successful.

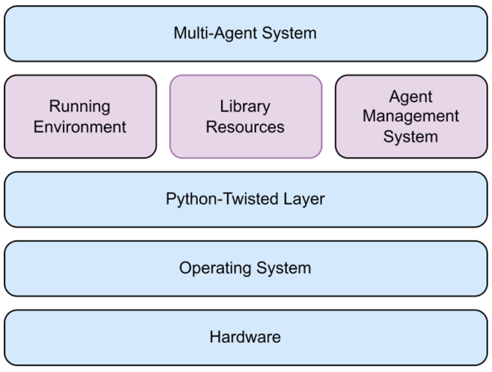


FIGURE 2
PADE architecture.

Therefore, if an agent requires communication with another agent, it will refer to the address of the target agent in its own table without asking the AMS agent. These agents can communicate even if the AMS agent is deactivated. The AMS agent can also record message interactions so that all agents in the network will send a copy of the message received to AMS each time.

AMS is the most significant network agent responsible for agent registration, monitoring, updating active agent tables, logging information exchanged among agents, sending orders to change, and deactivating agent's behavior.

An agent creation has a well-defined pattern of behavioral classes defined by PADE, protocol classes, and information from the class agent provided by core. Therefore, it is common to have an example in PADE that represents a given agent, and many classes required behavior examples representation defined by each agent. Figure 5 illustrates the chat behavior in PADE. Here, the chat agent class inherits all necessary features to execute and identify the agent to communicate with other counterparts. The chat port inherits all necessary behaviors from the chat class according to FIPA standards. Moreover, the chat class has the base behavior class to implement the base startup with methods.

3.3 Environmentally reinforcement learning (E-RL)

The environmentally reinforcement learning (E-RL) aims to provide a structured environment for LLM-based agent, enhancing their ability to perform code error correction and testing effectively. Through E-RL,

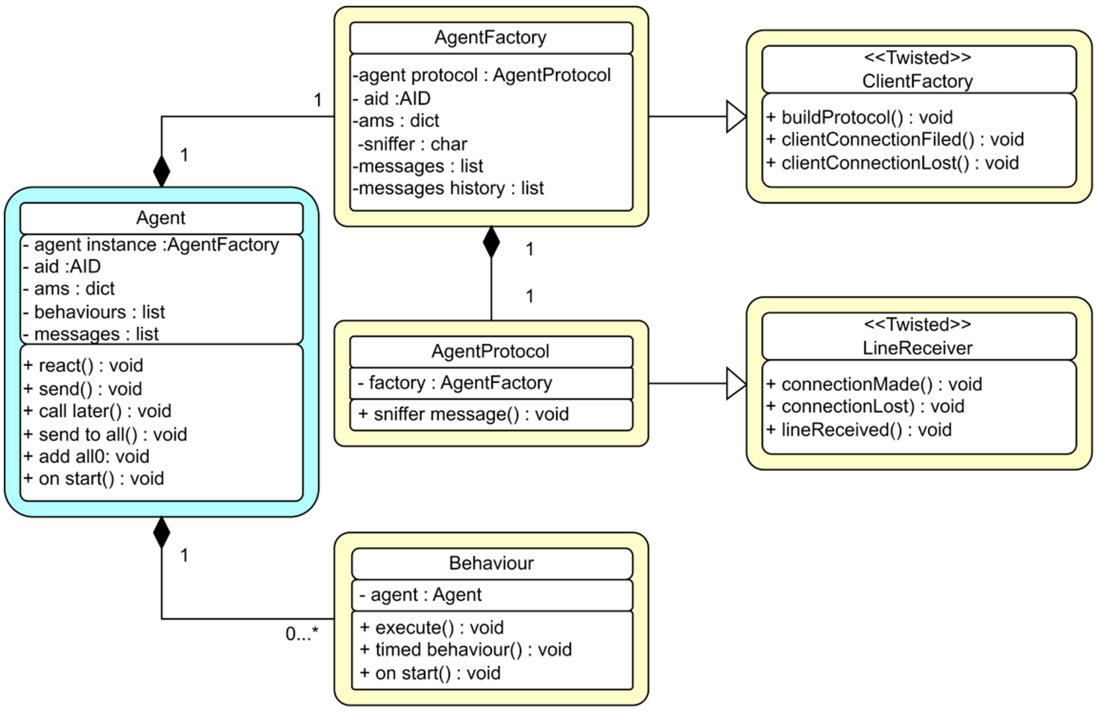


FIGURE 3
UML standard for classes in PADE framework.

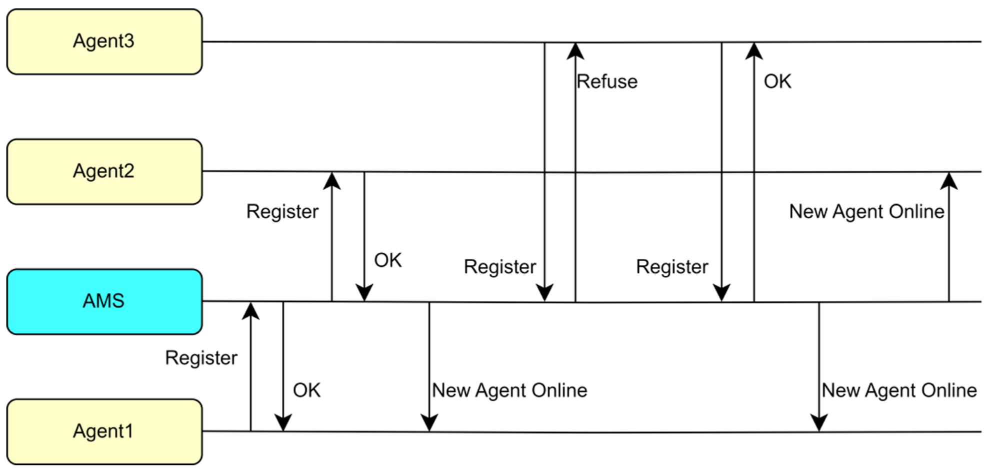


FIGURE 4
Registration and update of information in an example agent platform.

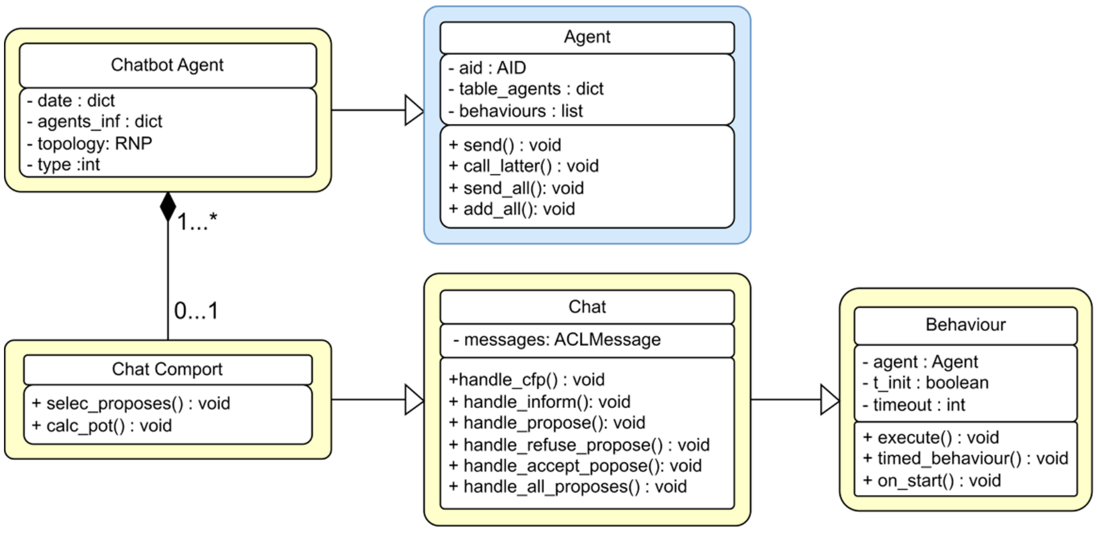


FIGURE 5
UML standard for a chat agent with FIPA chat behavior.

LLM-based agents select actions based on the current state S_{currect} and historical context H_{history} to perform tasks such as code error correction and interpretation. The interaction begins with a user-supplied code task description T_{task} and error code C_{error}, which together form the initial state S_{initial} of the system as shown in Equation 1:

$$S_{\text{initial}} = \left(T_{\text{task}}, C_{\text{error}}\right) \tag{1}$$

The LLM-based agent first explains the reason for the code error, E_{error} , based on the current state and historical context, as follows Equation 2:

$$E_{\text{error}} = f_{\text{explain}}\left(S_{\text{initial}}, H_{\text{history}}\right) \tag{2}$$

Next, the agent attempts to correct the erroneous code, C_{correct}, based on the generated explanation E_{error} and the current state as shown in Equation 3:

$$C_{\text{correct}} = f_{\text{correct}}\left(E_{\text{error}}, S_{\text{initial}}\right) \tag{3}$$

Once the corrected code is generated, it is executed and compared to two test cases with different levels of difficulty, denoted $S_{\text{test}\mathbf{1}}$ and $S_{\text{test}\mathbf{2}}$. The correctness of the generated code is evaluated as as shown in Equation 4:

$$R_{\text{test}\,1} = f_{\text{test}}\left(C_{\text{correct}}, S_{\text{test}\,1}\right), R_{\text{test}\,2} = f_{\text{test}}\left(C_{\text{correct}}, S_{\text{test}\,2}\right) \tag{4}$$

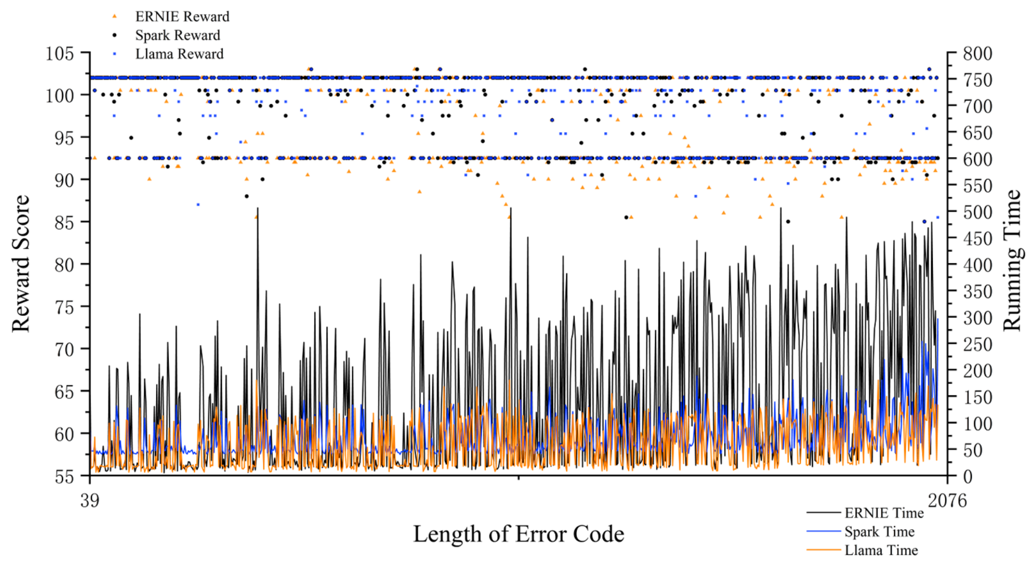


FIGURE 6
Reward scores and running time for different length of error code correction of three LLMs.

If both tests pass (i.e., $R_{\text{test}\,\mathbf{1}} = 1$ and $R_{\text{test}\,\mathbf{2}} = 1$), the interaction ends, and the final corrected code is returned as $C_{\text{final}} = C_{\text{correct}}$. If either of the tests fails, the generated code has errors and will be passed to the interpretation agent for re-evaluation, and the process continues. The process forms an iterative loop, where the code correction process continues until the generated code passes all tests or the maximum number of iterations $N_{\max}$ is reached.

The discrete state space S in E-RL consists of historical dialogue records that encapsulate the interactions between the user and LLM-based agents, which include task descriptions, error hints, and code corrections as shown in Equation 5:

$$S = \left\{S_{\text{initial}}, S_{\text{currect}}, \ldots, S_{\text{final}}\right\} \tag{5}$$

The action space A consists of all possible actions that the agent can take, which are represented by natural language outputs generated by the agent based on the current state, such as code corrections C_{correct}, error explanations E_{error}, or code annotations $A_{\text{annotation}}$. We define two different reward mechanisms at the same time.

The first reward mechanism evaluates the LLM-based agent's performance during interaction. Specifically, if the agent passes the basic test sample $S_{\text{test}\,\mathbf{1}}$, it will be rewarded with 2 points, and if it passes the more difficult test $S_{\text{test}\,\mathbf{2}}$, it will be rewarded with 3 points. Conversely, if the agent fails the tests, it will be penalized with 0.5 and 0.2 points, respectively. The reward function can be expressed as shown in Equation 6:

$$\mathrm{R}_{\text{success}} = \begin{cases} 2, \text{if fails basic test} \left(\mathrm{R}_{\text{test}\,1} = 1\right) \\ 3, \text{if passes difficult test} \left(\mathrm{R}_{\text{test}\,2} = 1\right) \end{cases}$$

$$\mathrm{R}_{\text{fail}} = \begin{cases} -0.5, \text{if fails basic test} \left(\mathrm{R}_{\text{test}\,1} = 0\right) \\ -0.2, \text{if fails difficult test} \left(\mathrm{R}_{\text{test}\,2} = 0\right) \end{cases} \tag{6}$$

Obtaining the results in Figure 6 based on the first reward mechanism, ERNIE performs the best in terms of performance but runs the slowest, LLAMA is rated as medium in terms of performance and runs at average speed, and Spark shows the worst results although it runs the fastest. Based on these different performance characteristics, we dynamically select the initial large model based on the length of the input code. Depending on the code length, we assigned short codes to Spark, medium-length codes to LLAMA, and longer codes to ERNIE, respectively.

The second reward mechanism aims to dynamically select the applicable language model based on the performance of the LLM-based agent under the code error correction dataset. We first use *softmax* functions to obtain the weights of the three large language models on the metrics of time and reward, respectively as shown in Equation 7:

$$\omega_{\text{time}} = \frac{e^{T_i}}{\sum_{i=1}^{k=3} e^{T_i}}, \omega_{\text{reward}} = \frac{e^{R_i}}{\sum_{i=1}^{k=3} e^{R_i}} \tag{7}$$

Where T_i is the execution time of each language model (ERNIE, LLAMA2-8B, and Spark V3) and R_i represents the reward score for each model based on its performance in terms of code error correction tasks. The softmax function transforms the execution time and reward into normalized values, which are used as relative weights to adjust the performance of each LLM in the final decision-making process. The weights ω_{time} and ω_{reward} can be interpreted as ratios that reflect the relative importance of execution time and reward, respectively.

The reward mechanism then calculates a composite score S_{LLM} for each language model based on code length L_{length}, run time ω_{reward}, reward value R, number of loops n (Table 1), and stability of the language model $S_{\text{stability}}$.

These metrics reflect the performance and operational status of the LLM-based agent during the interaction. Then, based on the calculated composite score, the language model with the highest score is selected as the

TABLE 1 Number of loops required for code error correction of three LLMs.

LLM	1 loop	2 loops	3 loops	4 loops	5 loops
ERNIE 4.0	337	60	26	14	265
Llama 3-8b	317	81	32	21	251
Spark V3	319	48	14	4	317

```
Input: code, error codes to be changed
run_time, number of executed loops
Output: LLM, selected LLM
Parameters: Recent_LLM, current use of LLM
LLM_performance, performance parameters LLMs
Parameter_weight, weight of parameters
1: /*Calculate the score for each LLM based on the current environmental state*/
2: for model, performance in LLM_performance.items() do
3:     Len_weight = code_length * Parameter_weight["length"] * LLM_performance["reward"]
4:     Reward_weight = Parameter_weight["reward"] * LLM_performance["reward"]
5:     time_weight =  Parameter_weight["time"] * LLM_performance["time"]
6:     run_time_weight =  Parameter_weight["run_time"] * run * LLM_performance["reward"]
7:     /*Stability penalty if the selected LLM same as the current environmental LLM*/
8:     if model == Recent_LLM then
9:         scores[model] = (Len_weight + Reward_weight - time_weight - run_time_weight) *
                       LLM_performance["stability"]
10:    else:
11:        scores[model] = Len_weight + Reward_weight - time_weight - run_time_weight
12: end for
13: /*Update the LLM used by the environment*/
14: LLM = max(scores, key = scores.get)
```

ALGORITHM 1
LLM options update.

main language model in the current environment. In this way, the reward mechanism is able to automatically adjust the selected language model according to the actual performance of the LLM-based agent to improve the efficiency and accuracy of code error correction and interpretation. The algorithm used by E-RL to select LLM is showed in Algorithm 1.

4 Experiments

4.1 Data description

The dataset used in the experiments is based on the Mostly Basic Programming Problems (MBPP) test set compiled by Austin et al. (2021), which consists of Python programming problems paired with various prompting words for code generation and corresponding automated test cases. To generate an error code dataset for the Co-Learning approach, we used ERNIE-3.5-8K (Sun et al., 2021), an early model with relatively low performance compared to the models employed in Co-Learning. The ERNIE-3.5-8 K model was used to generate erroneous code based on the provided prompting words. Although the generated code is not executable, it contains correct function names and relevant comments about the code logic, making it useful for subsequent experiments. This process resulted in the creation of a new dataset consisting of 702 error codes, test samples, and challenging test cases, as detailed in Table 2.

Figure 7 presents histograms illustrating the distribution of error code lengths in the dataset. It is instrumental in selecting the best LLM for the environment at the first-time code correction. By categorizing the length of the input message, E-RL will lead Co-Learning to select more capable LLMs for the next agent when obtaining long sentences. An attempt is made to reduce the number of error correction loops while sacrificing a proportion of the generation time, thus reducing the total time consumption and avoiding unnecessary computational costs.

4.2 Baseline LLM

ERNIE-4.0-8 K-0329 (Sun et al., 2021), Spark Desk V3 (iFLYTEK, 2023), and Meta-Llama-3-8b (Touvron et al., 2023) are selected as the open-source LLMs. Baseline LLMs are merged into the PADE multi-agent environment. E-RL selects the optimal model for the Co-Learning framework from the three to provide high-quality responses.

4.3 Experimental environment

The experiments are conducted on a server with a Xeon(R) Gold 5218 CPU @ 2.30GHz (16 cores). Models are implemented in PyTorch 2.1.0 with CUDA 12.1.

4.4 Main results

This experiment uses the original error code dataset, sets the maximum number of cycles to 5 (exceeding the number of cycles will

TABLE 2 Subset of the dataset samples.

Error code	Test list	Challenge test list
def remove_Occ(string, character): … …	['assert remove_Occ("hello","l") == "heo"', 'assert remove_Occ("abcda","a") == "bcd"', 'assert remove_Occ("PHP","P") == "H"']	['assert remove_Occ("hellolloll","l") == "helollol"', 'assert remove_Occ("","l") == ""']
def is_woodall(number): … …	['assert is_woodall(383) == True', 'assert is_woodall(254) == False', 'assert is_woodall(200) == False']	['assert is_woodall(32212254719) == True', 'assert is_woodall(32212254718) == False', 'assert is_woodall(159) == True']

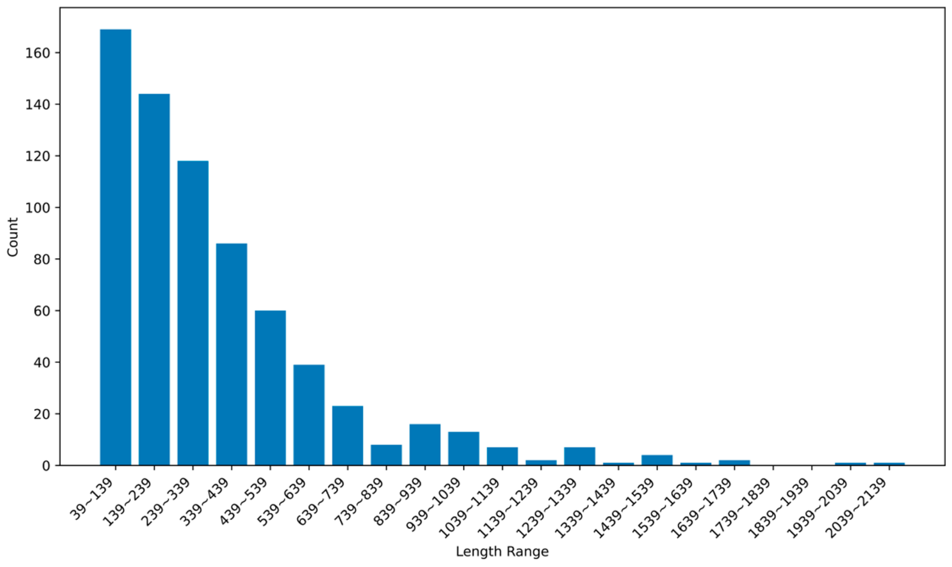


FIGURE 7
Distribution of error code length ranges.

directly determine the operation failure), and limits the memory length to three dialogue pairs to avoid exceeding the LLM single input message length limit.

Table 3 shows the number of successfully corrected loops, average running time, and final accuracy in the code correction task using a single LLM or E-RL and a collaborative learning framework based on multiple LLMs. Co-Learning mimicked rubber-duck debugging operations can be observed to help the model retry generation when the first generation goes wrong, with Llama 3-8b being the biggest beneficiary of the single LLM model, with 134 successful re-generations of the correct code.

Through E-RL, the probability of success of Co-Learning based on rubber duck testing is greatly increased; even if some first-time success probability is lost, a total of 196 examples are correctly modified due to E-RL. E-RL's contribution to the runtime is also undeniable, being only slower than the high-speed, low-enabled Spark V3, with an average test time of 99.8 s. Finally, Co-Learning with E-RL has the highest review success rate, reaching 67.80%.

4.5 Case study

Figure 8 depicts the actual situation of code error correction through Co-Learning. First, the master agent schedules the error correction task, and E-RL selects Llama as the initial LLM for joint learning based on the input message. The correct agent uses Llama to make initial modifications to error codes. Based on the results, it can be concluded that Llama generated the correct answer but secretly changed the function name. This phenomenon is obvious in the three LLMs. The LLM may choose more appropriate function names for the code based on the code content, while ignoring the user's needs.

The test agent detects that the generated code fails the test and returns an error message to the main agent. The main agent assigns code interpretation tasks in the hope of simulating rubber duck testing. The interpretation agent interprets the generated code and stores the contents in the main agent memory. E-RL re-selects Spark as the next agent's LLM, and the correct agent then re-corrects the code. Spark misinterprets the code as outputting multiples of the current number between 1 and 10, which may be related to the fact that the error returned by the test agent contains information about the first test sample, where the maximum number to be generated is 10.

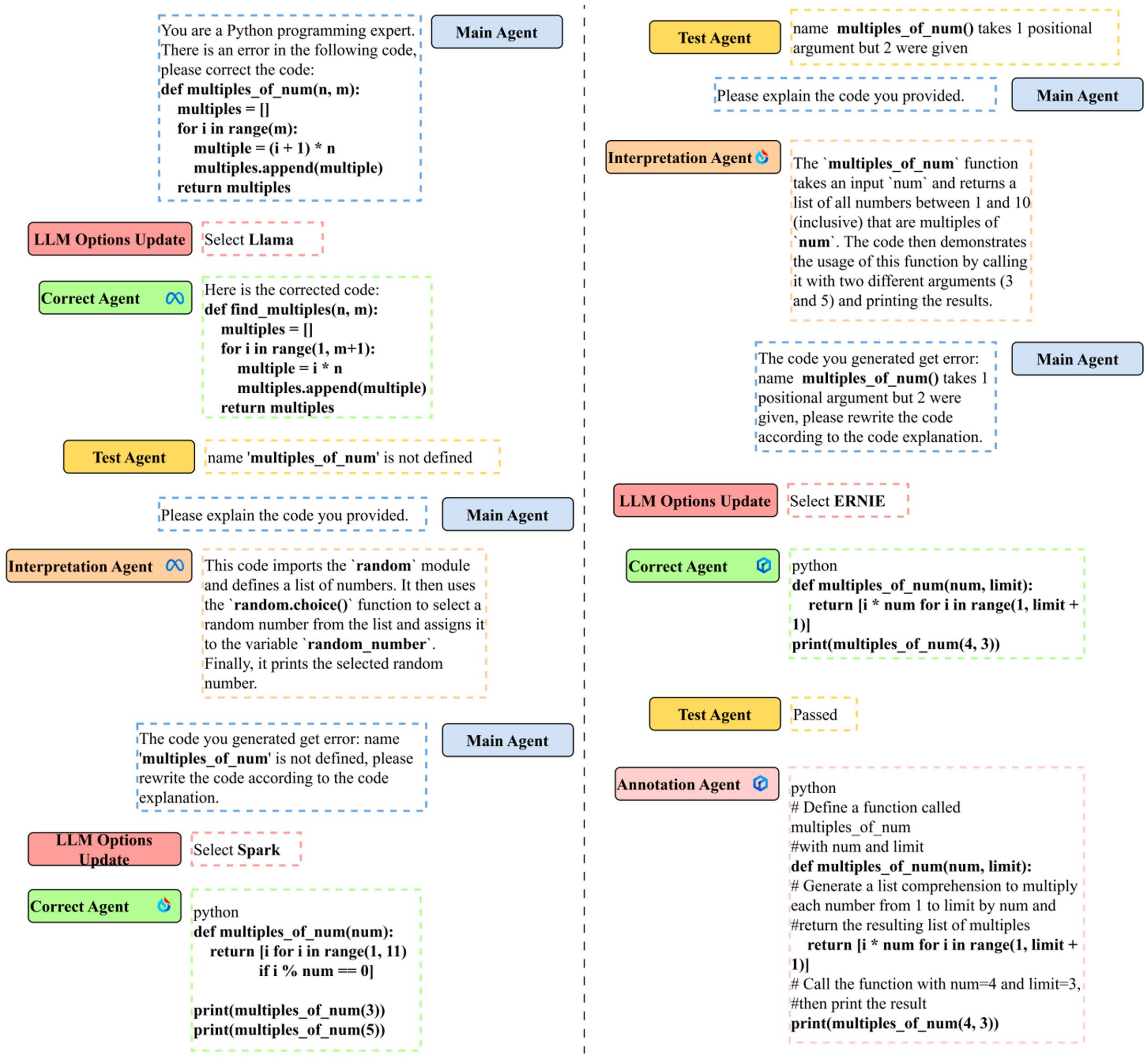
The test agent detected the error, and the leading main agent decide to make another rubber duck debugging. Then, LLM was changed to the most powerful ERNIE, which eventually generated the correct code, and the annotation agent added comments to the output.

This case shows how Co-Learning can continuously correct, understand, and then correct erroneous code by imitating the error correction process of human programmers, and generate information-intensive integrated code that only senior programmers can generate. With E-RL, Co-Learning attempts to balance model power and speed in the hope of generating the best response in the shortest possible time, creating code that is shorter and more refined than expected correct code, while using less time than a single model.

TABLE 3 Co-Learning with different LLM performance comparison.

Method	1 loop	2 loops	3 loops	4 loops	5 loops	Average running time (s)	Accuracy (%)
Co-Learning (ERNIE 4.0)	**337**	60	31	**29**	245	137.5	65.09
Co-Learning (Llama 3-8b)	317	81	32	21	251	112.8	64.24
Co-Learning (Spark V3)	319	48	14	4	**317**	**57.7**	54.84
Co-Learning (E-RL)	280	**104**	**65**	27	226	99.8	**67.80**

For 1-5 loops bold values represent the method that achieves the most correct results after n loops. For average running times and accuracy, bold values represent the method that achieves the best results.


Main Agent: You are a Python programming expert. There is an error in the following code, please correct the code:
```
def multiples_of_num(n, m):
    multiples = []
    for i in range(m):
        multiple = (i + 1) * n
        multiples.append(multiple)
    return multiples
```
LLM Options Update: Select Llama

Correct Agent: Here is the corrected code:
```
def find_multiples(n, m):
    multiples = []
    for i in range(1, m+1):
        multiple = i * n
        multiples.append(multiple)
    return multiples
```
Test Agent: name '**multiples_of_num**' is not defined

Main Agent: Please explain the code you provided.

Interpretation Agent: This code imports the `**random**` module and defines a list of numbers. It then uses the `**random.choice()**` function to select a random number from the list and assigns it to the variable `**random_number**`. Finally, it prints the selected random number.

Main Agent: The code you generated get error: name '**multiples_of_num**' is not defined, please rewrite the code according to the code explanation.

LLM Options Update: Select Spark

Correct Agent:
```
python
def multiples_of_num(num):
    return [i for i in range(1, 11)
            if i % num == 0]

print(multiples_of_num(3))
print(multiples_of_num(5))
```
Test Agent: name **multiples_of_num()** takes 1 positional argument but 2 were given

Main Agent: Please explain the code you provided.

Interpretation Agent: The `**multiples_of_num**` function takes an input `num` and returns a list of all numbers between 1 and 10 (inclusive) that are multiples of `**num**`. The code then demonstrates the usage of this function by calling it with two different arguments (3 and 5) and printing the results.

Main Agent: The code you generated get error: name **multiples_of_num()** takes 1 positional argument but 2 were given, please rewrite the code according to the code explanation.

LLM Options Update: Select ERNIE

Correct Agent:
```
python
def multiples_of_num(num, limit):
    return [i * num for i in range(1, limit +
1)]
print(multiples_of_num(4, 3))
```
Test Agent: Passed

Annotation Agent:
```
python
# Define a function called
multiples_of_num
#with num and limit
def multiples_of_num(num, limit):
# Generate a list comprehension to multiply
each number from 1 to limit by num and
#return the resulting list of multiples
    return [i * num for i in range(1, limit +
1)]
# Call the function with num=4 and limit=3,
#then print the result
print(multiples_of_num(4, 3))
```

FIGURE 8
Example of Co-Learning code correction.

5 Conclusion and future works

This study focuses on developing a code learning community (aka Co-Learning) framework based on an LLM-based multi-agent framework that leverages ambient reinforcement learning (E-RL) for agent self-improvement. The community aims to interpret error codes and perform code correction tasks to provide users with a more intelligent and personalized programming learning experience. Experiments show that the Co-Learning framework can effectively improve the code error correction capabilities of current LLM. E-RL dynamically determines the state of the environment and changes the selection of the LLM, which can speed up the code correction process and achieve significant improvements in the quality of the generated output.

In the future, Co-Learning will focus on further optimizing the E-RL algorithm to improve the agent's learning efficiency and performance. At present, it seems too simplistic to select E-RL parameters based only on model capabilities. Making Co-Learning's environmental reinforcement learning have dynamic self-updating weights by combining machine learning will be one of the main goals in the future; Expect to explore more complex tasks and scenarios, including error correction and code understanding in larger code bases, as well as code learning in different programming languages and domains.

Data availability statement

The raw data supporting the conclusions of this article will be made available by the authors, without undue reservation.

Author contributions

JY: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Software, Writing – original draft. YW: Data curation, Software, Validation, Writing – original draft. YZ: Investigation, Methodology, Visualization, Writing – original draft. WG: Investigation, Methodology, Visualization, Writing – original draft. ZX: Investigation, Methodology, Visualization, Writing – original draft. RL: Funding acquisition, Project administration, Resources, Supervision, Writing – review & editing.

Funding

The author(s) declare that no financial support was received for the research and/or publication of this article.

Acknowledgments

The authors thank the Beijing Normal University-Hong Kong Baptist University United International College and the IRADS lab for the provision for computer facility for the conduct of this research.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

References

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., et al. (2021). Program synthesis with large language models. *arXiv* [Preprint]. *arXiv:2108.07732*.

Barrault, L., Chung, Y. A., Meglioli, M. C., Dale, D., Dong, N., Duquenne, P. A., et al. (2023). SeamlessM4T-massively multilingual & multimodal machine translation. *arXiv* [Preprint]. *arXiv:2308.11596*.

Bellifemine, F., Poggi, A., and Rimassa, G. (2001). Developing multi-agent systems with JADE. In Intelligent agents VII agent theories architectures and languages: 7th international workshop, ATAL 2000 Boston, MA, USA, July 7–9, 2000 proceedings 7 (pp. 89–103). Springer, Berlin Heidelberg.

Bradshaw, J. M. (1997). An introduction to software agents. *Software Agents* 4, 3–46. Available at: https://scholar.google.com.hk/citations?user=M5fazi4AAAAJ&hl=zh-CN&oi=sra

Chen, X., Lin, M., Schärli, N., and Zhou, D. (2023). Teaching large language models to self-debug. *arXiv* [Preprint]. *arXiv:2304.05128*.

Fang, R., Bindu, R., Gupta, A., and Kang, D. (2024). LLM agents can autonomously exploit one-day vulnerabilities. *arXiv* [Preprint]. *arXiv:2404.08144*.

Ganguli, D., Askell, A., Schiefer, N., Liao, T. I., Lukošiūtė, K., Chen, A., et al. (2023). The capacity for moral self-correction in large language models. *arXiv* [Preprint]. *arXiv:2302.07459*.

iFLYTEK. (2023). Spark Desk V3.0. Version 1.0. Available online at: https://console.xfyun.cn/services/bm3 (Accessed November 9, 2023).

Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: a survey. *J. Artif. Intell. Res.* 4, 237–285. doi: 10.1613/jair.301

Kiran, B. R., Sobh, I., Talpaert, V., Mannion, P., Al Sallab, A. A., Yogamani, S., et al. (2021). Deep reinforcement learning for autonomous driving: a survey. *IEEE Trans. Intell. Transp. Syst.* 23, 4909–4926. doi: 10.1109/TITS.2021.3054625

Lee, R. S. (2005). Fuzzy-neuro approach to agent applications: From the AI perspective to modern ontology. Berlin: Springer Science & Business Media.

Lee, R. S., and Liu, J. N. (2001). I Jade Stock predictor – an intelligent multi-agent based time series stock prediction system. In intelligent agent technology: Research and Development (pp. 495–499).

Li, H., Hao, Y., Zhai, Y., and Qian, Z. (2023). The Hitchhiker's guide to program analysis: a journey with large language models. *arXiv* [Preprint]. *arXiv:2308.00245*.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., et al. (2015). Continuous control with deep reinforcement learning. *arXiv* [Preprint]. *arXiv:1509.02971*.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Clark, P., et al. (2024). Self-refine: Iterative refinement with self-feedback. *Adv Neural Inf Process Syst.* 36.

Melo, L. S., Sampaio, R. F., Leão, R. P. S., Barroso, G. C., and Bezerra, J. R. (2019). Python-based multi-agent platform for application on power grids. *Int. Trans. Electr. Energy Syst.* 29:e12012. doi: 10.1002/2050-7038.12012

Nair, V., Schumacher, E., Tso, G., and Kannan, A. (2023). DERA: enhancing large language model completions with dialog-enabled resolving agents. *arXiv* [Preprint]. *arXiv:2303.17071*.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., et al. (2022). Codegen: an open large language model for code with multi-turn program synthesis. *arXiv* [Preprint]. *arXiv:2203.13474*.

OpenAI (2023). Gpt-4 technical report. arxiv 2303.08774. View in Article, 2(5).

Schick, T., and Schütze, H. (2020). Exploiting cloze questions for few shot text classification and natural language inference. *arXiv* [Preprint]. *arXiv:2001.07676*.

Shinn, N., Labash, B., and Gopinath, A. (2023). Reflexion: an autonomous agent with dynamic memory and self-reflection. *arXiv* [Preprint]. *arXiv:2303.11366*.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., et al. (2016). Mastering the game of go with deep neural networks and tree search. *Nature* 529, 484–489. doi: 10.1038/nature16961

Sumers, T. R., Yao, S., Narasimhan, K., and Griffiths, T. L. (2023). Cognitive architectures for language agents. *arXiv* [Preprint]. *arXiv:2309.02427*.

Sun, Y., Wang, S., Feng, S., Ding, S., Pang, C., Shang, J., et al. (2021). Ernie 3.0: large-scale knowledge enhanced pre-training for language understanding and generation. *arXiv* [Preprint]. *arXiv:2107.02137*.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., et al. (2023). Llama 2: open foundation and fine-tuned chat models. *arXiv* [Preprint]. *arXiv:2307.09288*.

Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., et al. (2024). A survey on large language model based autonomous agents. *Front. Comp. Sci.* 18, 1–26. doi: 10.1007/s11704-024-40231-1

Xi, Z., Chen, W., Guo, X., He, W., Ding, Y., Hong, B., et al. (2023). The rise and potential of large language model based agents: a survey. *arXiv* [Preprint]. *arXiv:2309.07864*.

Zhang, H., Du, W., Shan, J., Zhou, Q., Du, Y., Tenenbaum, J. B., et al. (2023). Building cooperative embodied agents modularly with large language models. *arXiv* [Preprint]. *arXiv:2307.02485*.

Zhou, W., Jiang, Y. E., Li, L., Wu, J., Wang, T., Qiu, S., et al. (2023). Agents: an open-source framework for autonomous language agents. *arXiv* [Preprint]. *arXiv:2309.07870*.