

SELF-EVOLVING MULTI-AGENT NETWORKS FOR SOFTWARE DEVELOPMENT

Anonymous authors

Paper under double-blind review

ABSTRACT

LLM-driven multi-agent collaboration (MAC) systems have demonstrated impressive capabilities in automatic software development at the function level. However, their heavy reliance on human design limits their adaptability to the diverse demands of real-world software development. To address this limitation, we introduce EvoMAC, a novel self-evolving paradigm for MAC networks. Inspired by traditional neural network training, EvoMAC obtains text-based environmental feedback by verifying the MAC network’s output against a target proxy and leverages a novel textual backpropagation to update the network. To extend coding capabilities beyond function-level tasks to more challenging software-level development, we further propose RSD-Bench, a requirement-oriented software development benchmark, which features complex and diverse software requirements along with automatic evaluation of requirement correctness. Our experiments show that: i) The automatic requirement-aware evaluation in RSD-Bench closely aligns with human evaluations, validating its reliability as a software-level coding benchmark. ii) EvoMAC outperforms previous SOTA methods on both the software-level RSD-Bench and the function-level HumanEval benchmarks, reflecting its superior coding capabilities.

1 INTRODUCTION

Automatic software development focuses on generating code from natural language requirements. Code is a universal problem-solving tool, and this automation presents significant potential to provide substantial benefits across all areas of our lives (15). Recently, the industry has introduced several large language model (LLM)-driven coding assistants, including Microsoft’s Copilot (20), Amazon’s CodeWhisperer (1), and Google’s Codey (7). These coding assistants significantly advance human efficiency and yield considerable commercial benefits. Despite the initial success of LLMs in assisting with line-level coding, they struggle to tackle more complex coding tasks. This limitation stems from the restricted reasoning abilities of single LLMs and their lack of capacity for long-context understanding (25; 13; 26).

To handle function-level coding tasks, numerous multiple language agent collaboration (MAC) systems have been proposed (8; 4; 10; 31; 16; 21). These MAC systems function as LLM-driven agentic workflow. They follow human-designed standardized operating procedures to divide the complex coding tasks into simpler subtasks within the workflow, allowing each agent to conquer specific subtasks. These MAC systems significantly advance coding capabilities from line-level to function-level tasks. However, current MAC systems rely on heuristic designs. These human-crafted static systems have two inherent limitations: i) their performance is confined to human initialization. Given the diversity of real-world coding tasks, human design cannot fully address the specific needs of each task; and ii) they lack the flexibility to adapt to new tasks. This rigidity necessitates that researchers and developers manually decompose tasks and create prompts. The complexity of this process inhibits effective human optimization for adapting to new challenges.

To address these limitations, we present EvoMAC, a novel self-evolving paradigm for MAC networks. EvoMAC’s key feature is its ability to iteratively adapt both agents and their connections during test time for each task. Inspired from the standard neural network training, the core idea of self-evolution is to obtain text-based environmental feedback by verifying the MAC network’s generation against a target proxy, then leverage a novel textual back-propagation to update the MAC network. Following this general paradigm, we specify EvoMAC for software development, which comprises

three essential components: i) an adaptable MAC network-based coding team that generates code through feed-forward; ii) a specifically designed testing team that creates unit test cases serving as the target proxy and verifies the generated code in the compiler to produce objective feedback; and iii) an updating team that uses the textual back-propagation algorithm to update the coding team. By cycling these three components, the coding team can iteratively evolve and generate codes that are better aligned with the unit test cases, eventually fulfilling more requirements of the coding task.

Our self-evolving MAC network has the potential to further advance coding capabilities from function-level to more complex software-level tasks. As it can iteratively address lengthier task requirements and cater to realistic software development demands. However, existing benchmarks typically focus on specific individual functions (5; 3; 30; 12) or bug-fixing (11), leaving a significant gap in providing comprehensive requirements for software development. This gap makes it difficult to fully assess the potential of our self-evolving MAC network.

To support the development of software-level coding capabilities, we propose RSD-Bench, a novel requirement-oriented software development benchmark. It is the first benchmark that features both complex and diverse software requirements, as well as the automatic evaluation of requirement correctness. RSD-Bench involves 53 coding tasks with 616 requirements, covering two typical software types, Website, and Game, and two requirement difficulty levels, Basic and Advanced. Each coding task consists of two components: i) multiple requirements that clearly outline measurable software functionalities, item by item, and ii) paired black-box test cases that automatically verify the correctness of each requirement. RSD-Bench can achieve automatic evaluation with these synchronized pairs of requirements and test cases. The RSD-Bench introduces new software-level challenges, including lengthy requirement analysis and long-context coding, which are essential in real-world software development but are absent in existing benchmarks.

To validate the effectiveness of our proposed EvoMAC and RSD-Bench, we conduct three key evaluations. First, we compare our automatic evaluation in RSD-Bench with human evaluation, achieving a coherence score of 99.22%, demonstrating its reliability. Second, we compare EvoMAC against five multi-agent and three single-agent baselines. EvoMAC significantly outperforms previous SOTAs by 26.48%, 34.78%, and 6.10% on Website Basic, Game Basic, and HumanEval, respectively, underscoring its effectiveness. Third, we evaluate EvoMAC with varying evolving times and two different driving LLMs. The results indicate that EvoMAC consistently improves with more evolving times and shows convincing enhancements regardless of the driving LLM used, further demonstrating the effectiveness of our self-evolving design.

To sum up, our contributions are:

- We propose EvoMAC, a novel self-evolving MAC network, and apply it to software development. EvoMAC can iteratively adapt both agents and their connections during test time for each task.
- We propose RSD-Bench, a novel requirement-oriented software development benchmark. It is the first benchmark that features both complex and diverse software requirements, as well as the automatic evaluation of requirement correctness.
- We conduct comprehensive experiments and validate that: automatic evaluation in RSD-Bench is highly aligned with human evaluation; EvoMAC outperforms previous SOTAs, and self-evolving promises continuous improvement with evolving times.

2 RELATED WORKS

LLM-based multi-agent collaboration. LLM-driven multi-agent collaboration (MAC) systems (29; 9; 34; 27; 8; 4; 18) enable multiple agents to share information and collaboratively complete the overall task. These MAC systems function as agentic workflows. They have demonstrated enhanced problem-solving capabilities in various domains, such as mathematics (10), software development (23; 8), embodied task (19) and social simulation (34; 22; 14). However, these systems (27; 6) heavily rely on manually designed workflows, which lack generalizability and the labor-intensive nature of manual design poses significant limitations. To address this issue, we propose a novel self-evolving paradigm, which allows agents to update and improve through external feedback, enabling dynamic adaptation and more advanced performance across varied tasks.

Software development benchmarks. Software development benchmarks aim to evaluate models in the task of generating code from natural language descriptions (32). These benchmarks typically include task definitions and evaluation criteria. Existing benchmarks can be categorized into three

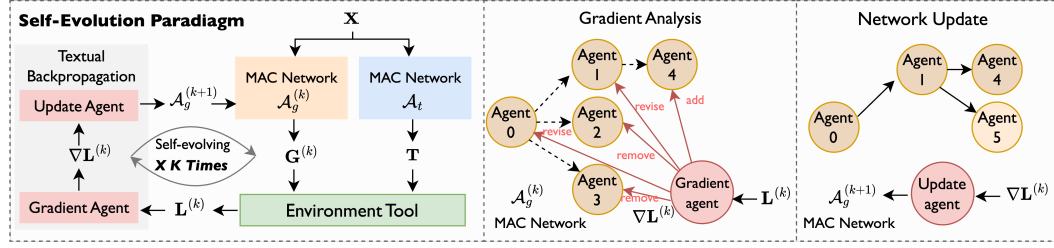


Figure 1: The general self-evolving paradigm.

types: i) function completion (HumanEval (5), MBPP (3), EvalPlus (17), xCodeEval (12)); ii) bug repair (SWE-bench (11)); and iii) software generation (SRDD (23), SoftwareDev (8)). Function completion and bug repair benchmarks are confined to function-level task definitions, missing the diverse realistic software requirements. Software generation benchmarks often depend on expensive human evaluations or indirect similarity-based measurements, unable to automatically and accurately verify the requirement correctness. To address these limitations, we introduce RSD-Bench, the first benchmark contains both diverse software requirements and automatic evaluation of requirement correctness. It can support the development of more realistic software-level coding capabilities.

3 EVOMAC: SELF-EVOLVING MULTI-AGENT COLLABORATION NETWORK

This section presents *EvOMAC*, a novel self-evolving multi-agent collaboration network and its application to software development. The key feature of *EvOMAC* is its ability to iteratively adapt both agents and their connections during test-time for each task, mimicking the back-propagation process, a core algorithm in neural network training. We first formulate a general self-evolving paradigm in Sec. 3.1 and then describe its application to software development in Sec. 3.2.

3.1 A GENERAL SELF-EVOLVING PARADIGM VIA TEXTUAL BACKPROPAGATION

Multi-agent collaboration network. A multi-agent collaboration (MAC) network is a computational graph representing agentic workflows, where multiple agents empowered by LLMs interact as interconnected nodes to coordinate and share information for complex task-solving. The intuition behind to divide the complex task into more specific and manageable subtasks for each agent, allowing the overall task to be gradually conquered through the agentic workflow. Mathematically, we represent a MAC network with N autonomous agents as a directed acyclic graph $\mathcal{A} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_i\}_{i=1}^N$ is the set of N nodes, and $\mathcal{E} = \{e_{i,j}\}_{i,j \in [1, \dots, N], i \neq j}$ is the set of directed edges with no circles. The i -th node v_i represents the i -th autonomous agent with the prompt p_i , which specifies its subtask. The edge $e_{i,j}$ represents the task dependency between the i -th agent and the j -th agent, indicating that the j -th agent's subtask should be executed after the i -th agent's subtask in the agentic workflow. The overall graph topology specifies the agentic workflow. Analogy to traditional neural networks, agents function similarly to neurons, with agent prompts serving as neurons' weights and the agentic workflow as the layers and connections.

The feed-forward pass of MAC network is the execution of the agentic workflow. In this process, each agent is given two inputs: the initial task requirement and the output from the previous agent. Using these, each agent produces an output that fulfills its specific subtask. Eventually, the last agent's generation constitutes the final output, integrating all completed subtasks. Note that the initial task requirement is input to each agent as context, providing supplementary details to aid in the implementation of each subtask.

Recently, various MAC networks have been designed using human expertise to assign fixed agent prompts and workflows (8; 4), resembling untrained neural networks. However, these designs solely rely on human priors and lack adaptability, causing limited performance improvement over a single agent. To overcome this, inspired by neural network training, we propose a self-evolving paradigm for multi-agent collaboration networks, enabling both agents and their connections to dynamically evolve during test-time for each given task.

Optimization problem. Here we consider a general generation task. During test-time, given a task, the MAC network performs a feed-forward pass to generate the final output without knowing its quality. The key to evolution during test-time is to set up a target proxy for the MAC network to guide its improvements in the generated output. Here we consider this target proxy as the conditions for task completion, such as unit tests in coding, and we can produce such a target proxy by another

group of autonomous agents based on the same task description. Then, the quality of each generated output can be verified according to the target proxy. This approach relies on two key assumptions: (i) generating a target proxy is significantly simpler than completing the original generation task, and (ii) the generated output can be correctly verified against the target proxy through an objective environment. These assumptions are practical in many applications. For example, in code generation, producing unit tests, the expected input-output pairs, is much easier than generating the entire code; meanwhile, a code compiler naturally acts as the objective environment to check the correctness of the generated code against the unit test, providing objective and informative feedback.

Mathematically, let \mathbf{X} be the textual description of a task. Given the MAC network \mathcal{A}_g , the generated output is $\mathbf{G} = \Phi(\mathbf{X}, \mathcal{A}_g)$, where $\Phi(\cdot, \cdot)$ is the general feed-forward operator that executes the agentic workflow, processing the input text through the MAC network. Similarly, the target proxy is $\mathbf{T} = \Phi(\mathbf{X}, \mathcal{A}_t)$, where \mathcal{A}_t is another MAC network designed for producing the target proxy. Note that we aim to evolve and optimize \mathcal{A}_g , while keep \mathcal{A}_t predefined and fixed. The optimization of our self-evolution is formulated as,

$$\mathcal{A}_g^* = \min_{\mathcal{A}_g} \langle \Phi(\mathbf{X}, \mathcal{A}_g), \mathbf{T} \rangle_E, \quad \text{subject to: } \mathbf{T} = \Phi(\mathbf{X}, \mathcal{A}_t), \quad (1)$$

where $\langle \cdot, \cdot \rangle_E$ is an objective environment executor that receives the generated output and the target proxy as inputs and outputs a text-based environmental feedback. Akin to the loss function in traditional neural network training, which quantifies the difference between the generated output and the ground-truth, the objective in equation 1 evaluates whether the generated output meets the conditions of the task completion using the environment, subsequently producing execution reports as the text-based environmental feedback. Here the minimization operation \min is defined to reduce the failures during execution. With the guidance of the target proxy and the objective feedback given by the environment, the MAC network can improve its success rate of task completion during test-time.

Note that, another straightforward way to enable the MAC network’s evolution is through the self-critique strategy (33; 24; 28; 2), which employs a critique agent to assess the generated output directly. This approach has two inherent limitations: i) the critique may be subjective and biased, and ii) the critique agent can have hallucinations, causing inconsistencies and errors. These limitations can cause the MAC network to become entrenched in its own preferences or evolve in the wrong direction, especially iterating multiple times; see our experimental validations in Tab. 2. In comparison, our approach leverages an environment executor to provide objective feedback, preventing bias and hallucinations.

While we use the analogy between our self-evolution process and neural network training for motivating, they are significantly different in three key aspects: (i) our self-evolution occurs at test time without a dedicated training phase; (ii) it evolves for each specific task individually rather than over a batch of samples; and (iii) the environmental feedback are usually texts, not be numerical values, which cannot be optimized by the standard backpropagation. This motivates us to propose our textual backpropagation.

Solution based on textual backpropagation. The self-evolution solution iteratively updates the MAC network using a textual backpropagation algorithm, guided by the environmental feedback. The core idea is to analyze the influence of each agent in the MAC network \mathcal{A}_g to the final environmental feedback and use these analyses to update the agent prompts and the agentic workflow in \mathcal{A}_g . This is achieved by two collaborative agents, each responsible for one of the two key steps: (i) textual gradient analysis and (ii) network updates.

First, the gradient agent takes the environmental feedback as the input and outputs textual gradients that describe the impact of each agent in the MAC network. Let $\mathcal{A}_g^{(k)}$ and $\mathbf{L}^{(k)}$ be the MAC network and the environment feedback at the k -th iteration. The textual gradient is then $\nabla \mathbf{L}^{(k)} = \mathcal{G}(\mathcal{A}_g^{(k)}, \mathbf{L}^{(k)})$, where $\mathcal{G}(\cdot, \cdot)$ is the gradient analysis operator managed by the gradient agent; see its prompt in Appendix. The textual gradient details three-fold information for each agent inside $\mathcal{A}_g^{(k)}$: 1) whether this agent’s subtask is fulfilled; 2) whether this agent introduces errors; and 3) whether any subtask is missed in the current MAC network.

Second, based on the textual gradients, the updating agent iterates the MAC network as $\mathcal{A}_g^{(k+1)} = \mathcal{U}(\mathcal{A}_g^{(k)}, \nabla \mathbf{L}^{(k)})$, where $\mathcal{U}(\cdot, \cdot)$ is the updating operator managed by the updating agent. This operator guides the updates from three-folds: 1) removing the agents whose subtasks have been completed; 2)

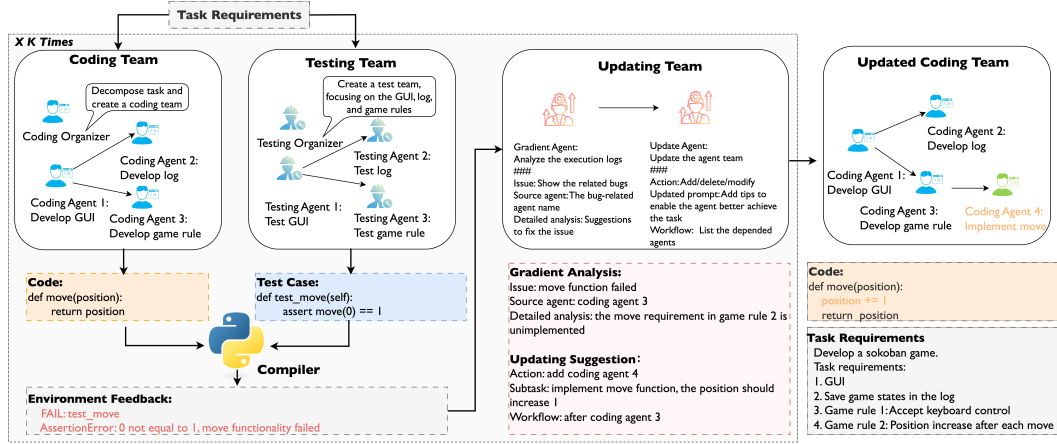


Figure 2: EvoMAC takes task requirements as input and iteratively updates the coding team to generate code that better fulfills the requirements.

revising the erroneous agent’s prompts by adding potential solutions provided in the gradient analysis; and 3) adding new agents for missing subtasks and restructuring the workflows based on the subtask dependencies noted in the gradient analysis; see the prompt details in Appendix. These adjustments address existing issues and fulfill unmet requirements in the current generation of the MAC network, promising improvements in the updated version.

Note that, the key of the textual backpropagation is the prompt designs for both gradient analysis and network updates. The design must i) thoroughly evaluate the subtask of each agent in the MAC network according to the objective environment feedback and determine necessary adjustments to the MAC network to address existing issues, fulfilling the unmet requirements; and ii) maintain coherence, ensuring that issues identified by the gradient agent can be effectively resolved by the updating agent’s modifications to the MAC network.

3.2 SELF-EVOLUTION FOR SOFTWARE DEVELOPMENT

In this section, we apply the self-evolving paradigm to the task of software development. The overall architecture of the proposed self-evolving multi-agent collaboration network for software development is illustrated in Fig. 1. Given a coding task, the coding team, corresponding to the MAC network \mathcal{A}_g , generates all the codes through its forward-pass; the testing team, associated with the MAC network \mathcal{A}_t , is responsible for creating the target proxy; that is, unit tests of the coding task; and the objective environment tool is realized through the compiler. The identified bugs during execution form the textual environmental feedback. The updating team, consisting of two collaborative agents, manages the textual backpropagation. By continuously cycling through feed-forward, feedback collection and textual backpropagation processes, the coding team is iteratively refined to more closely align with the test cases.

Since unit test generation is much easier than the original logical code generation, the testing team usually can produce high-quality test cases, which are closely aligned with the task requirements. Then, improving alignment with the unit tests through MAC network updates ensures better adherence to the actual task requirements.

Coding team for feed-forward. In the feed-forward process, the coding team synthesizes code according to the given coding task. To handle the extensive software requirements, the coding team is implemented as a MAC network. It divides the comprehensive requirements into a sequence of smaller, more specific function implementation subtasks, and progressively conquers them through the agentic workflow. Unlike existing MAC systems that heuristically decompose coding tasks and define the agentic workflow, we initialize the MAC network using a novel self-organizing approach. A coding organizer agent automatically and flexibly decomposes the task requirements into subtasks and assembles the coding agent team accordingly. The number of coding agents is dynamic, adjusting in response to the task requirements. Note that, the quality of the generated code is unknown during the forward pass, which necessitates the self-evolving paradigm to iteratively refine the generation.

Testing team and compiler for feedback collection. To verify whether the generated code meets the requirements of the coding task, we employ unit tests as the target proxy. These test cases consist

of input-output pairs tailored to specific requirements. For example, a test case for a keyboard control requirement would detail the type of control as the input and describe the expected behavior as the output. To create flexible and comprehensive unit tests, we set up the testing team as a MAC network and also initialize it in a self-organized way. A testing organizer agent automatically decomposes our specified key testing criteria into subtasks and accordingly forms the testing agent team.

Once the test cases and generated code are ready, they are executed in the compiler, which functions as the environmental tool, producing execution logs. These logs clearly point out the gap between the generated code and the test cases. It shows satisfied testing requirements, existing function errors, and unmet testing requirements. This feedback information can be used to verify whether each agent’s subtask is accomplished and guide the MAC network update.

Updating team for textual back-propagation. The updating team consists of two collaborative agents: the gradient agent and the updating agent, adjusting the MAC network based on the execution logs, including the agent prompts and workflows. This process consists of two steps. First, the gradient agent summarizes the textual gradient by identifying accomplished subtasks for satisfied requirements, appending new subtasks for unmet requirements, and analyzing errors to detail their originating agents and revising suggestions. Second, the updating agent modifies the coding agent team by removing agents that have completed their subtasks, adding new agents for the new subtasks, and revising agent prompts to address issues identified in the previous generation. The agent workflow is updated once the agent team is revised, based on the dependencies among the subtasks.

4 RSD-BENCH: REQUIREMENT-ORIENTED SOFTWARE DEVELOPMENT BENCHMARK

This section introduces RSD-Bench, a requirement-oriented benchmark designed to assess the ability of models to handle software-level coding tasks. Each coding task involves multiple detailed software requirements. These requirements specify each functionality and constraint of the software, item by item, serving as measurable benchmarks for assessing the software’s effectiveness. As shown in Fig. 3, unlike previous instruction-oriented approaches (23; 8) which rely on brief instructions as input, RSD-Bench uses comprehensive software requirements as input, complemented by unit test cases to automatically evaluate the correctness. This benchmark provides software-level coding tasks and automatic evaluation, aligning more closely with real-world software development practices.

4.1 BENCHMARK CONSTRUCTION

RSD-Bench involves two typical real-world software types: game and website. They can reflect different coding capacities demanded in realistic software development. Game often requires handling dynamic interactions, real-time state changes, and user-driven operations, focusing on elements like logic execution, initialization, and game state transitions. Website emphasizes static and dynamic content management, user interaction through forms and buttons, and ensuring page elements are displayed and functional. RSD-Bench involves diverse requirements, each paired with a test case. Specifically, RSD-Bench provides 53 unique coding tasks and 616 test cases. For details on generating software requirements and test cases, see Appendix.

RSD-Bench introduces two requirement difficulty levels, including basic and advanced, to reflect the varying complexity of real-world software development tasks. The basics reflect the fundamental and more achievable requirements, such as interaction, control, and logging. The advanced reflects more complex software functionalities, such as game logical rules, and dynamic web content management. The details can be referred to the appendix.

4.2 AUTOMATIC EVALUATION

RSD-Bench supports automatic evaluation of requirement correctness. It achieves this by pairing a specifically designed black-box test case with each requirement. The test case can directly verify whether the generated code achieved the requirement. Its evaluation metric is the accuracy, which quantifies the proportion of correctly passed test cases. It is similar to the `pass@1` metric in HumanEval (5), which evaluates the pass ratio of correctly achieved functions against the total functions via unit test verification. It is a fully automated evaluation process, eliminating the need for human involvement while still providing accurate and reliable assessments.

Previous benchmarks for software code generation mainly rely on two evaluation methods. One method is human evaluation (8), which is time-consuming and not scalable for large datasets. The other method is indirect evaluations (23), which defines metrics like consistency, completeness, and

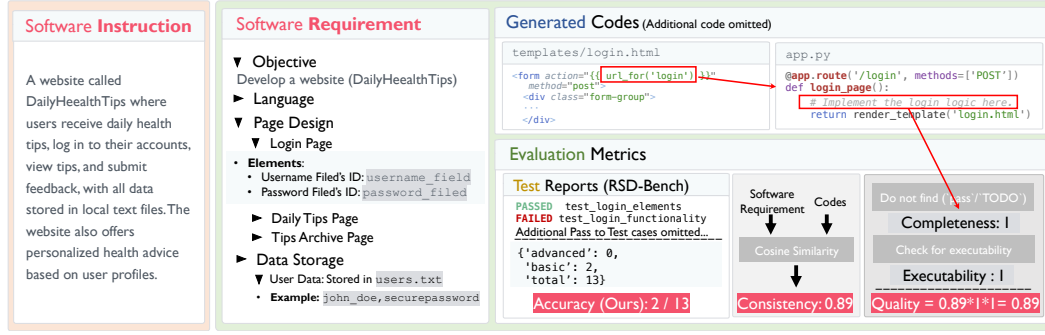


Figure 3: Comparison between instruction-oriented and requirement-oriented evaluations. RSD-Bench accurately reflects requirement fulfillment with the proposed accuracy score of 2/13, while the indirect evaluation misjudges with high scores (0.89), failing to detect missing functionality.

quality. Consistency measures how closely the generated software aligns with the original requirement description by comparing the cosine similarity between the two. Completeness is determined by detecting the presence of placeholder (such as `pass` or `TODO`), which results in a binary value of 0 or 1. Quality is then calculated as the product of several factors: consistency, completeness, and executability. As illustrated in Fig. 3, they could not measure the correctness of the generated code in fulfilling requirements. In contrast, RSD-Bench’s test cases-based evaluation is more rigorous and precise. These test cases can accurately verify the correctness of generated code in fulfilling the requirements. RSD-Bench promises reliable and scalable automatic evaluation. In the experiments, we have validated the significant advantages of the proposed automatic evaluation over the previous metrics, including consistency and quality; see Fig. 4.

4.3 FEATURES

Challenging and diverse software requirements. RSD-Bench features long-context software requirements (averaging 507/1011 words for game and website tasks, respectively), unlike instruction-oriented benchmarks (5; 3; 11) that rely on brief prompts. These detailed requirements better reflect real-world lengthy and complex software development challenges.

Requirement-aware precise and efficient evaluation. RSD-Bench employs detailed software requirements and automated unit tests to precisely measure how well generated software meets its objectives. Generated codes are evaluated based on pass rates from running specific test cases, offering an accurate and efficient process. In contrast, instruction-oriented benchmarks rely on brief prompts, which lack constraints and make evaluation less reliable, often requiring labor-intensive or indirect evaluation.

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

Baselines. To validate the effectiveness of our EvoMAC, we conducted comparisons against both single-agent and multi-agent baselines. The single-agent baselines involve three prominent large models: GPT-4o-Mini (gpt-4o-mini), Claude-3.5-Sonnet (claude-3-5-sonnet-20240620), and Gemini (gemini-1.5-flash). For multi-agent baselines, we included five state-of-the-art (SOTA) methods: MetaGPT (8), Autogen (27), Mapcoder (10), Agentverse (6), and ChatDev (23). To ensure a fair comparison, all multi-agent baselines, including our EvoMAC, are powered by the efficient and powerful GPT-4o-Mini model. Additionally, to demonstrate the adaptability and robustness of our EvoMAC, we developed two EvoMAC variants using GPT-4o-Mini and Claude-3.5-Sonnet.

Datasets. Our experiments cover both the proposed RSD-Bench and the standard coding benchmark HumanEval (5). HumanEval comprises 164 Python function completion problems, where the task is to generate code from a single function description.

5.2 EFFECTIVENESS OF RSD-BENCH’S EVALUATION AND EVOMAC

RSD-Bench’s automatic evaluation metric (accuracy) is highly aligned with human evaluation. Our primary goal is to validate the effectiveness of the proposed automatic evaluation in RSD-Bench by comparing it with two existing evaluation metrics: consistency and quality, both from SRDD (23).

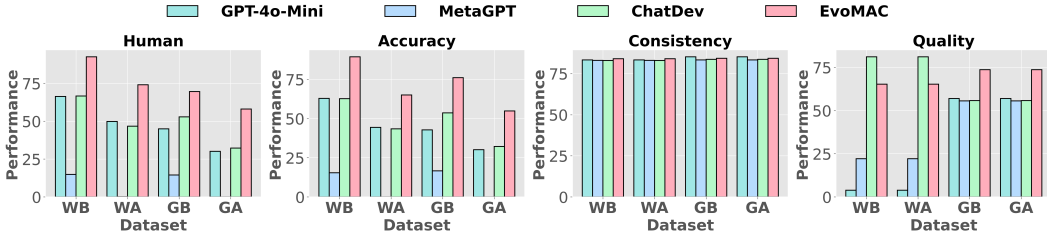


Figure 4: Performance of four methods in terms of four evaluation metrics, including human evaluation, our automatic evaluation (accuracy), consistency, and quality. WB and WA represent Web Basic and Web Advanced respectively. GB and GA represent Game Basic and Game Advanced respectively. Our accuracy metric is highly aligned with human evaluation across four dataset settings.

Table 1: Comparison of EvoMAC with five multi-agent and three single-agent SOTA baselines, all powered by GPT-4o-Mini. **Red** values represent the percentage improvement of EvoMAC, shade in pink, over the single-agent baselines, shade in grey.

Method	Model	RSD-Bench				HumanEval
		Website(%)		Game(%)		(%)
		Basic	Advanced	Basic	Advanced	Pass@1
Single Agent	Gemini-1.5-Flash	29.79±1.00	11.61±2.34	21.74±6.39	6.45±6.97	73.17
	Claude-3.5-Sonnet	58.90±1.48	37.11±1.06	44.20±5.41	18.29±13.26	89.02
	GPT-4o-Mini	62.90±2.52	44.40±4.21	42.76±15.50	30.10±11.87	88.41
Multi Agent	MetaGPT	15.41±0.00	0.00±0.00	16.67±2.71	0.00±0.00	88.41
	Autogen	25.68±4.14	5.40±3.34	17.39±1.78	0.00±0.00	85.36
	MapCoder	34.70±1.59	14.57±0.66	29.71±6.72	7.52±6.10	90.85
	Agentverse	15.41±0.00	0.00±0.00	37.67±8.20	16.13±4.55	90.85
	ChatDev	62.67±0.28	43.45±0.77	53.63±5.70	32.26±4.55	70.73
	EvoMAC	89.38±1.01	65.05±1.56	77.54±2.04	51.60±4.54	94.51
		+26.48	+20.65	+34.78	+21.50	+6.10

For a fair comparison, our golden standard is human evaluation, conducted by two expert code engineers who manually verify the fulfillment of requirements by interacting with the developed software. This process is tedious, taking around four hours per expert to evaluate the entire benchmark. The effectiveness of an evaluation metric depends on how closely it aligns with human evaluation.

Fig. 4 presents the performance of four methods in terms of four evaluation metrics, including human evaluation, our automatic evaluation, consistency, and quality. We see that: i) our automatic evaluation is highly aligned with human evaluation across two software types (Website and Game), four methods, (GPT-4o-Mini, MetaGPT, ChatDev, and our EvoMAC), and two requirement difficulties (Basic and Advanced). The correlation coefficient between human evaluation and our accuracy metric is 0.9922, demonstrating the effectiveness of the proposed automatic evaluation in RSD-Bench; ii) Consistency and quality metrics differ significantly from human evaluation, with correlation coefficients of 0.2583 and 0.3041, respectively. This discrepancy occurs because consistency in SRDD measures similarity, and quality in SRDD focuses on executability, which does not guarantee that all requirements are met. This highlights the need for RSD-Bench, as the SRDD benchmark does not support requirement-oriented software development.

EvoMAC outperforms previous SOTAs on both software-level and function-level coding benchmarks: RSD-Bench and HumanEval. Tab. 1 compares EvoMAC with five multi-agent and three single-agent SOTA baselines, all powered by GPT-4o-Mini for a fair comparison. We see that EvoMAC significantly outperforms previous SOTAs across all datasets. EvoMAC outperforms single-agent methods by 26.48% on the RSD-Bench Website Basic and 34.78% on the RSD-Bench Game Basic, as well as surpassing existing multi-agent methods by over 20%. This highlights the effectiveness of multi-agent collaboration and the power of EvoMAC.

5.3 EFFECTIVENESS OF EVOLVING

Fig. 6 shows the accuracy of EvoMAC over multiple evolving iterations on the RSD-Bench and HumanEval. Each figure presents two curves: one for EvoMAC powered by GPT-4o-Mini (red) and the other by Claude-3.5 (blue). We have the following findings:

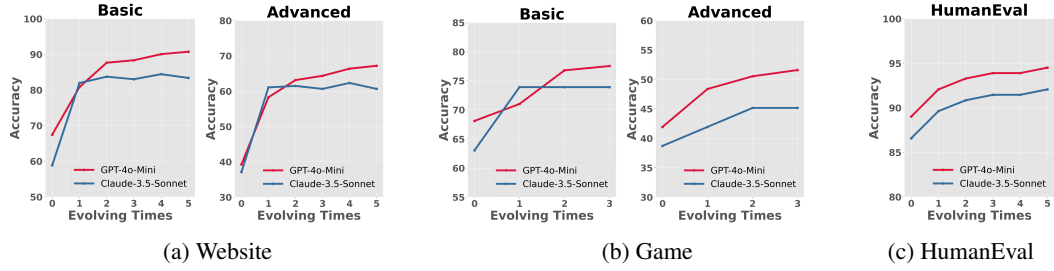


Figure 6: Effect of EvoMAC performance across evolving times empowered by GPT-4o-Mini and Claude-3.5-Sonnet on Website, Game, and HumanEval datasets. The figure shows EvoMAC continuously improves with the evolving times on both LLM drives.

Table 2: Ablation study about coding/testing team with single/multi-agent, with/without evolving, and with/without environment tool. Best performances are bolded.

	Coding	Testing	Evol.	Env.	Website(%)		Game(%)	
					Basic	Advanced	Basic	Advanced
a) Single	-	-	-	-	63.70	41.70	42.76	30.10
b) Multi	-	-	-	-	67.47	39.27	68.10	41.93
c) Single	Single	✓	✓	✓	80.82	60.32	71.73	41.93
d) Multi	Single	✓	✓	✓	83.90	60.72	76.08	41.93
e) Single	Multi	✓	✓	✓	83.56	61.94	73.91	45.16
f) Multi	Multi	✓	-	-	78.08	52.23	55.80	33.32
g) Multi	Multi	✓	✓	✓	90.75	67.20	77.54	51.60

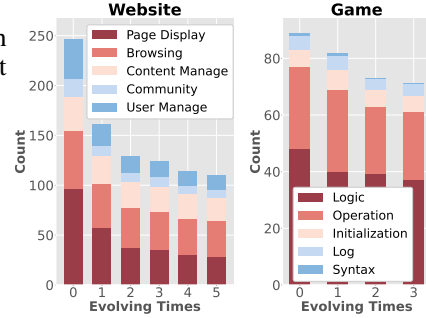


Figure 5: Failure case distribution across evolving times on Website and Game.

EvoMAC continuously improves with the evolving times. Fig. 6 shows that as evolving iterations increase, performance consistently improves across all five dataset settings, covering two difficulty levels, two software types, and both requirement-oriented and function complement benchmarks. This highlights the effectiveness, generalizability, and robustness of the self-evolving approach, encouraging EvoMAC to evolve whenever possible.

EvoMAC indistinguishably improves with different driving LLM. From Fig. 6, we see that: i) both EvoMAC variants continuously improve with evolving iterations, demonstrating the robustness of the self-evolving design; ii) the two curves do not intersect, indicating that the EvoMAC variant powered by a more powerful single model consistently outperforms the other, highlighting the advantage of using a stronger model. Success builds on success.

Failure case analysis. Fig. 5 shows the failure case statistics across iterations for Website and Game, showing a general decrease in errors as iterations progress. We see that: i) the most common errors are page display issues in Website and logic errors in Game; ii) page errors are resolved more quickly, while logic errors persist, suggesting that more isolated issues are easier to fix during the evolution process. This results in a sharp initial performance improvement as simpler problems are addressed early, followed by a plateau as more complex issues remain unresolved, shown in Fig. 6.

5.4 ABLATION STUDY

To assess the effectiveness of each component, Table 2 details an ablation study featuring seven EvoMAC variants.

Effectiveness of objective environment feedback. Environment feedback, such as code execution logs, is essential for software development. Variant f) omits this tool, instead using an LLM-driven agent to critique the code. Comparing Variant g) with Variant f) shows a notable performance drop: Website tasks decrease by 12.67% and 14.97%, and Game tasks by 21.74% and 18.28% for Basic and Advanced levels, respectively. This underscores the importance of objective environmental feedback, as agent-driven critiques may introduce bias and fail to guide the evolution effectively.

Effectiveness of multi-agent collaboration in coding team and testing team. Comparing Variant g) to Variant e), we observe a performance decrease of 7.19% and 5.26% on Website Basic and Advanced respectively, when the coding team is reduced to a single agent. Similarly, comparing Variant g) to Variant d), there is a performance drop of 6.85% and 6.48% on Website Basic and Advanced respectively, also when the team is reduced to a single agent. These results demonstrate

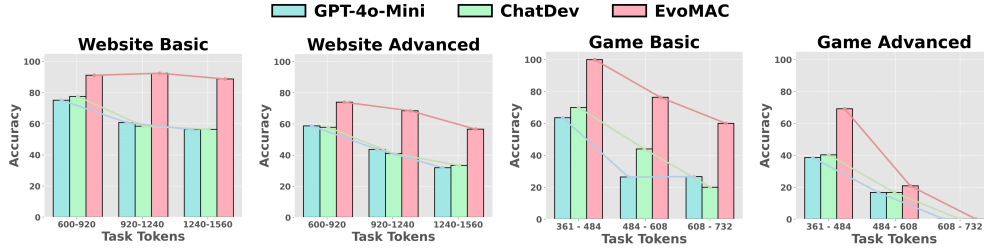


Figure 7: EvoMAC outperforms previous multi-agent and single-agent systems across all the context lengths across the four dataset settings on RSD-Bench.

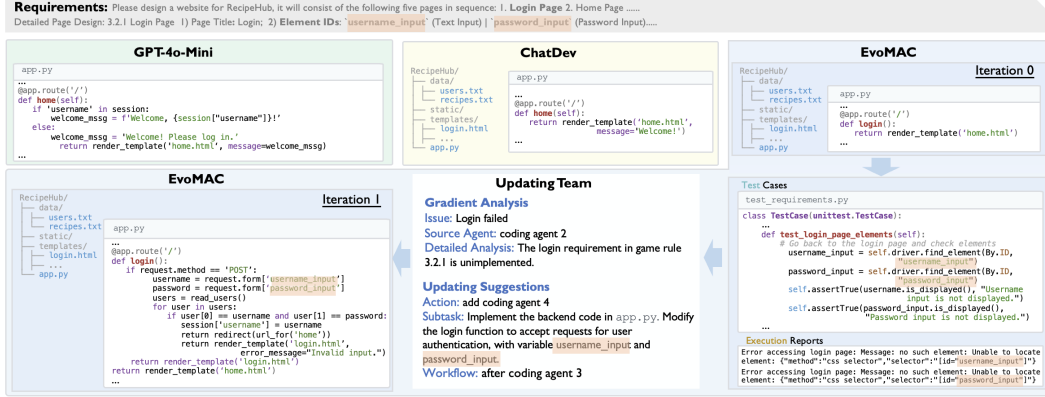


Figure 8: We show the generated code of single-agent, GPT-4o-Mini, and multi-agent systems, ChatDev, and our EvoMAC (iteration =0/1) given the Website task (RecipeHub). After evolving, EvoMAC can revise previous issues and fulfill the task requirement.

the necessary for involving multi-agent collaboration, highlighting that multi-agent setups offer more flexible adjustments and enhanced capabilities for evolution.

Effectiveness in handling varied task token lengths. Fig. 7 shows a comparison of task token lengths and performance across GPT-4o-Mini, ChatDev, and EvoMAC. We see that: i) EvoMAC consistently outperforms ChatDev and GPT-4o-Mini across all context lengths, with its self-evolving mechanism enabling the identification and correction of missed contexts and errors during iterations; ii) EvoMAC experiences less performance degradation on the RSD-Bench Website than on the Game, as Website tasks are more modular and can be broken into subtasks, whereas Game tasks require more coordinated management, making them more challenging.

5.5 CASE STUDY

Fig. 8 presents the generated code by a single agent, GPT-4o-Mini, multi-agent systems, ChatDev, and our EvoMAC before and after evolving (iteration=0/1). We see that: i) EvoMAC after evolving can correct issues from previous iterations and successfully fulfill the task requirements; ii) multi-agent systems tend to better comprehend the task requirements and produce more well-structured code.

6 CONCLUSION

We propose EvoMAC, a novel self-evolving paradigm for MAC networks. EvoMAC iteratively adapts agents and their connections during the testing phase of each task. It achieves this with a novel textual back-propagation algorithm. EvoMAC can push coding capabilities beyond function-level tasks and into more complex, software-level development. Furthermore, we propose RSD-Bench, a novel requirement-oriented software development benchmark. RSD-Bench features both complex and diverse software requirements, as well as the automatic evaluation of requirement correctness. Comprehensive experiments validate that the automatic requirement-aware evaluation in RSD-Bench aligns closely with human evaluation. EvoMAC outperforms previous SOTAs in both software-level RSD-Bench and function-level HumanEval benchmarks.

Future works. In the future, we plan to introduce a reward model to enhance the self-evolving paradigm’s ability to learn from feedback and extend the RSD-Bench to more software types.

REFERENCES

- [1] Amazon. CodeWhisperer. In <https://platform.qa.com/course/amazon-codewhisperer-generating-code-ai-4679/introduction>, 2022.
- [2] Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-rag: Learning to retrieve, generate, and critique through self-reflection. *ArXiv*, abs/2310.11511, 2023.
- [3] 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.
- [4] 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. In *The Twelfth International Conference on Learning Representations*.
- [5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. 2021.
- [6] Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chen Qian, Chi-Min Chan, Yujia Qin, Yaxi Lu, Ruobing Xie, et al. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors in agents. *arXiv preprint arXiv:2308.10848*, 2(4):6, 2023.
- [7] Google. Codey. In <https://console.cloud.google.com/vertex-ai/publishers/google/model-garden/codechat-bison>, 2023.
- [8] Sirui Hong, Xiwu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.
- [9] Wenyue Hua, Lizhou Fan, Lingyao Li, Kai Mei, Jianchao Ji, Yingqiang Ge, Libby Hemphill, and Yongfeng Zhang. War and peace (waragent): Large language model-based multi-agent simulation of world wars. *arXiv preprint arXiv:2311.17227*, 2023.
- [10] Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Mapcoder: Multi-agent code generation for competitive problem solving, 2024.
- [11] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2023.
- [12] Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval, 2023.
- [13] Jiaqi Li, Mengmeng Wang, Zilong Zheng, and Muhan Zhang. Loogle: Can long-context language models understand long contexts?, 2024.
- [14] Junkai Li, Siyu Wang, Meng Zhang, Weitao Li, Yunghwei Lai, Xinhui Kang, Weizhi Ma, and Yang Liu. Agent hospital: A simulacrum of hospital with evolvable medical agents, 2024.
- [15] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de, Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel Jaymin Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378:1092 – 1097, 2022.

- [16] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, December 2022.
- [17] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.
- [18] Zhao Mandi, Shreeya Jain, and Shuran Song. Roco: Dialectic multi-robot collaboration with large language models. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 286–299. IEEE, 2024.
- [19] Zhao Mandi, Shreeya Jain, and Shuran Song. Roco: Dialectic multi-robot collaboration with large language models. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 286–299. IEEE, 2024.
- [20] Microsoft. Copilot. In <https://www.microsoft.com/en-us/microsoft-copilot/meet-copilot>, 2023.
- [21] Anton Osika. GPT-Engineer. In <https://github.com/AntonOsika/gpt-engineer>, 2023.
- [22] Xianghe Pang, Shuo Tang, Rui Ye, Yuxin Xiong, Bolun Zhang, Yanfeng Wang, and Siheng Chen. Self-alignment of large language models via monopolylogue-based social scene simulation. In *Forty-first International Conference on Machine Learning*, 2024.
- [23] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. Chatdev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2023.
- [24] Karthik Valmeekam, Matthew Marquez, and Subbarao Kambhampati. Can large language models really improve by self-critiquing their own plans? *ArXiv*, abs/2310.08118, 2023.
- [25] Chonghua Wang, Haodong Duan, Songyang Zhang, Dahua Lin, and Kai Chen. Ada-leval: Evaluating long-context llms with length-adaptable benchmarks, 2024.
- [26] Xindi Wang, Mahsa Salmani, Parsa Omid, Xiangyu Ren, Mehdi Rezagholizadeh, and Armaghan Eshaghi. Beyond the limits: A survey of techniques to extend the context length in large language models, 2024.
- [27] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation, 2023.
- [28] Yifan Xu, Xiao Liu, Xinghan Liu, Zhenyu Hou, Yueyan Li, Xiaohan Zhang, Zihan Wang, Aohan Zeng, Zhengxiao Du, Wenyi Zhao, Jie Tang, and Yuxiao Dong. Chatglm-math: Improving math problem-solving in large language models with a self-critique pipeline. *ArXiv*, 2024.
- [29] Yuzhuang Xu, Shuo Wang, Peng Li, Fuwen Luo, Xiaolong Wang, Weidong Liu, and Yang Liu. Exploring large language models for communication games: An empirical study on werewolf. *arXiv preprint arXiv:2309.04658*, 2023.
- [30] Guang Yang, Yu Zhou, Xiang Chen, and Xiangyu Zhang. Codescore-r: An automated robustness metric for assessing the functionalcorrectness of code synthesis, 2024.
- [31] 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*, abs/2405.15793, 2024.
- [32] Zibin Zheng, Kai-Chun Ning, Jiachi Chen, Yanlin Wang, Wenqing Chen, Lianghong Guo, and Weicheng Wang. Towards an understanding of large language models in software engineering tasks. *ArXiv*, abs/2308.11396, 2023.
- [33] 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.
- [34] Caleb Ziems, William Held, Omar Shaikh, Jiaao Chen, Zhehao Zhang, and Diyi Yang. Can large language models transform computational social science? *Computational Linguistics*, 50(1):237–291, 2024.