Self-Organized Agents: A LLM Multi-Agent Framework toward Ultra Large-Scale Code Generation and Optimization

Yoichi Ishibashi
TsukushiAI
ishibashi.tsukushiai@gmail.com

Yoshimasa Nishimura
TsukushiAI
nishimura.tsukushiai@gmail.com

Abstract

Recent advancements in automatic code generation using large language model (LLM) agent have brought us closer to the future of automated software development. However, existing single-agent approaches face limitations in generating and improving large-scale, complex codebases due to constraints in context length. To tackle this challenge, we propose **S**elf-**O**rganized multi-**A**gent framework (**SoA**), a novel multi-agent framework that enables the scalable and efficient generation and optimization of large-scale code. In SoA, self-organized agents operate independently to generate and modify code components while seamlessly collaborating to construct the overall codebase. A key feature of our framework is the automatic multiplication of agents based on problem complexity, allowing for dynamic scalability. This enables the overall code volume to be increased indefinitely according to the number of agents, while the amount of code managed by each agent remains constant. We evaluate SoA on the HumanEval benchmark and demonstrate that, compared to a single-agent system, each agent in SoA handles significantly less code, yet the overall generated code is substantially greater. Moreover, SoA surpasses the powerful single-agent baseline by 5% in terms of Pass@1 accuracy. [1]

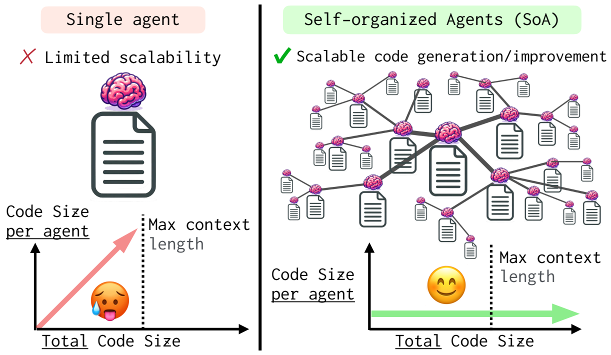


Figure 1: **Left (single agent):** A single agent is solely responsible for the entire implementation. As the codebase grows larger, the load increases for code generation, modification, and memory management, making it difficult to manage and develop. The larger the entire codebase becomes, the more it puts pressure on the context length during self-debugging, limiting the amount of code that can be managed. **Right (SoA):** The implementation is distributed among multiple agents. The agents are independent; code generation, modification, and memory management are separated from other agents. Each agent manages only its own part, allowing it to focus on the implementation regardless of the complexity of the entire codebase. Furthermore, agents automatically multiply according to the complexity of the problem. This allows for the generation and modification of complex and large-scale code while maintaining a constant amount of code management/generation/modification per agent.

1 Introduction

In recent years, research on agents using Large Language Models (LLMs) (Brown et al., 2020; OpenAI, 2023; Touvron et al., 2023), such as ReAct (Yao et al., 2023b), Reflexion (Shinn et al., 2023), Toolformer (Schick et al., 2023), and AutoGPT [2], has been expanding the possibilities of automating human tasks. These advancements have particularly contributed to the rapid development of automatic code generation techniques in the field of automated application and tool development (Hong et al., 2023; Dong et al., 2023; Huang et al., 2023). Compared to non-agent-based methods (Muennighoff et al., 2023; Li et al., 2023b), these research achievements have led to remarkable performance improvements in automatic code generation (Zhong et al., 2024; Zhou et al., 2023).

Most recent research has focused on single-agent approaches for code generation. These single-agent code generation methods face limitations, especially in terms of scalability, when the implemen-

[1] Our code will be available at https://github.com/tsukushiAI/self-organized-agent.

[2] https://github.com/Significant-Gravitas/

tation becomes complex and requires a large codebase. The main reason for this technical difficulty is that a single agent must manage the entire code generation process alone. For instance, implementing a machine learning algorithm involves several stages, such as data preprocessing, algorithm training, and result evaluation, which include many functions and classes. When these complex components are combined, the codebase inevitably becomes very large. However, there are limitations to the context length of LLMs, and as the number of input tokens increases, the inference performance decreases (Levy et al., 2024; Shaham et al., 2023; Li et al., 2023a). Consistently understanding and generating or modifying appropriate code for such an extensive codebase poses a significant challenge for a single agent in terms of comprehending and managing the context. Consequently, the single-agent approach struggles to efficiently generate and modify code as its complexity and size increase.

To tackle these challenges, we propose a self-organized multi agent framework that can automatically generate and modify large-scale code (Figure 1). *Self-organization* (Ashby, 1947) is a phenomenon in which living organisms or matter create an orderly, large structure as a result of their individual autonomous behaviors, despite lacking the ability to oversee the entire system. In our framework, self-organized agents, each responsible for different code parts or tasks, independently generate and modify code. With the self-organization of agents, a single agent no longer needs to comprehend the entire codebase, making it possible to scale up large-scale code simply by increasing the number of agents. Another feature of our framework is that agents automatically multiply according to the complexity of the problem, allowing the overall codebase to expand while keeping the amount of code handled by each agent constant. These features enable the dynamic and flexible generation and modification of large-scale code, which was impossible with the traditional single-agent approach.

In our experiments, we evaluated the performance of this framework using HumanEval (Chen et al., 2021), a benchmark for code generation. The results show that our self-organized multi-agent framework outperformed Reflexion (Shinn et al., 2023), an existing powerful code generation agent (§ 4.1), demonstrating the effectiveness of our approach in generating and modifying code. Furthermore, through a detailed analysis of the experimental results, we revealed how agents automatically multiply according to the complexity of the problem, effectively scaling up the overall code volume while keeping the code generation per agent constant (§ 4.2). These experimental results support the contribution of our framework, which overcomes the scalability issues faced by single-agent approaches and provides a solution capable of handling larger projects.

2 Code Generation Task

The code generation task involves generating Python functions from docstrings (Chen et al., 2021). In this task, an agent is given a docstring that defines the types of the function's inputs and expected outputs, as well as the specific requirements that the function should meet. The agent is then required to generate the code for a function that fulfills the specified functionality. The generated code is verified for accuracy using unit tests, and the quality of the code is evaluated based on its ability to pass the test cases. As with previous studies (Shinn et al., 2023; Zhong et al., 2024; Zhou et al., 2023), we use the evaluation metric Pass@1 (Chen et al., 2021), where a problem is considered solved if any of the k code samples pass all test cases.

3 Self-organized Agent Framework

Our Self-organized Agents (SoA) framework enables efficient implementation of large-scale and complex code by having self-organized agents independently generate and modify small-scale and simple code. In this section, we introduce the important components of SoA, namely the agents and the layers responsible for more abstract processing than the agents, and finally introduce the code generation and modification protocols in the SoA framework.

3.1 Child Agent

Child agents implement a given function based on its docstrings. As shown in Figure 2, this agent has a simple structure consisting of two elements: an LLM and memory. The LLM generates code from the given docstrings and modifies the code based on the results of unit tests. The memory stores the code generated by the agent itself and retrieves the latest code to be input to the LLM along with the unit test feedback during code modification. If an agent has these minimal specifications, it is possible to use an off-the-shelf agents (e.g., Reflexion) as a Child agent. We deliberately use a simple agent to verify the effectiveness of SoA in a simple setup.

Code Generation The main role of Child agents is to generate functions that meet the specifications based on the given function's docstrings. As shown in Figure 2, the agent follows the instructions to generate the rest of the function and complete it. The completed function implementation is stored in memory, and the unit tests for the function are also stored as they form the basis for future code modifications.

Code Modification: Empowering Child Agents with Self-Organization and Adaptability One of the most remarkable aspects of agents in the SoA framework is their ability to autonomously improve their code based on the state of nearby agents . This process sets SoA apart from traditional agent approaches and showcases the power of self-organization in code modification. While existing agents like Reflexion (Shinn et al., 2023) rely solely on the results of unit tests, Child agents in SoA go beyond this limitation by independently observing the state of their mother agent, such as differences in modifications and feedback. By gathering this invaluable information from their surrounding environment, Child agents can adapt their behavior and make more informed decisions about code modification, even without explicit instructions. The modifications and feedback generated by the Mother agent serve as an important source of information for the Child agents. Armed with these insights, Child agents can more effectively modify their own code, contributing to the overall improvement of the codebase in a way that is both efficient and adaptive. Figure 3 illustrates this process, which begins with the execution of unit tests and the retrieval of the latest implementation from memory. The Child agent then harnesses the power of the LLM to create a code modification proposal, seamlessly combining the information observed from the Mother agent with the test results and the latest implementation details. By storing the modified code in memory, Child agents create a feedback loop that continuously refines and improves the codebase over time. This iterative process, driven by the principles of self-organization and adaptability, enables SoA to tackle complex code modification tasks with efficiency and effectiveness. As Child agents work in harmony with their Mother agent, they contribute to the creation of a more optimized and large codebase.

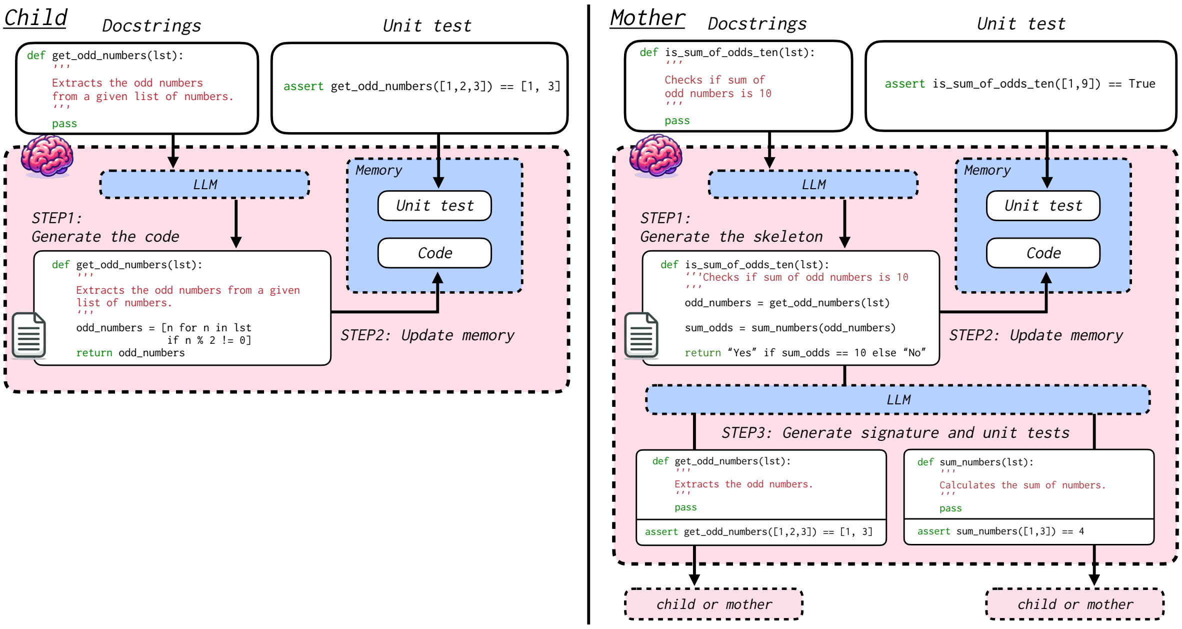
3.2 Mother Agent

The Mother is an agent that generates new agents (Mother or Child). Similar to Child agents, the Mother agent independently implements the specific Python function based on its given docstrings. The Mother has memory, code generation capabilities, and self-debugging functions, as same as Child agents. The unique feature of the Mother agent is its ability to generate multiple Child agents according to the complexity of the problem and delegate parts of the implementation to these agents. This structure allows the Mother agent to focus on implementing abstract processes, while the Child agents generated by the Mother agent concentrate on implementing concrete processes. This division of labor enhances the overall efficiency and flexibility of the SoA framework.

Code Generation We explain the code generation process by the Mother agent using the implementation example of the `is_sum_of_odds_ten` function shown in Figure 2. The starting point is the function's docstrings and unit tests, which are memorized for reference in the later self-debugging phase. The first task of the Mother agent is to generate a skeleton of the implementation from the given docstrings, including subfunctions such as `get_odd_numbers` to extract odd numbers and `sum_of_numbers` to calculate their sum. The number and types of these subfunctions are automatically determined by the LLM based on the complexity of the problem.

It is important to note that these subfunctions are unimplemented, and the Mother agent does not directly implement them. Instead, it delegates the implementation of the subfunctions to other agents, allowing the Mother agent to focus on generating the skeleton and streamline its own code generation process. After the docstrings and unit tests for the subfunctions are generated, they are assigned to newly initialized agents for implementation. These agents proceed with the implementation of their respective functions without looking at the internals of the `is_sum_of_odds_ten` function implemented by the Mother agent. Since agents within the same Mother can work asynchronously, the overall code generation process is streamlined.

Code Modification The Mother's code modification is almost the same as the Child's code modifi-



```
def get_odd_numbers(lst):
    '''
    Extracts the odd numbers
    from a given list of numbers.
    '''
    pass
```

```
assert get_odd_numbers([1,2,3]) == [1, 3]
```

```
def get_odd_numbers(lst):
    '''
    Extracts the odd numbers from a given
    list of numbers.
    '''
    odd_numbers = [n for n in lst
                   if n % 2 != 0]
    return odd_numbers
```

```
def is_sum_of_odds_ten(lst):
    '''
    Checks if sum of
    odd numbers is 10
    '''
    pass
```

```
assert is_sum_of_odds_ten([1,9]) == True
```

```
def is_sum_of_odds_ten(lst):
    '''Checks if sum of odd numbers is 10
    '''
    odd_numbers = get_odd_numbers(lst)

    sum_odds = sum_numbers(odd_numbers)

    return “Yes” if sum_odds == 10 else “No”
```

```
def get_odd_numbers(lst):
    '''
    Extracts the odd numbers.
    '''
    pass
```

```
assert get_odd_numbers([1,2,3]) == [1, 3]
```

```
def sum_numbers(lst):
    '''
    Calculates the sum of numbers.
    '''
    pass
```

```
assert sum_numbers([1,3]) == 4
```

Figure 2: Overview of code generation. Child agents generate executable Python function from a given docstring. The Mother agent generates the skeleton of the function. The Mother spawns a new initialized agent (Child or Mother) and delegates unimplemented functions.

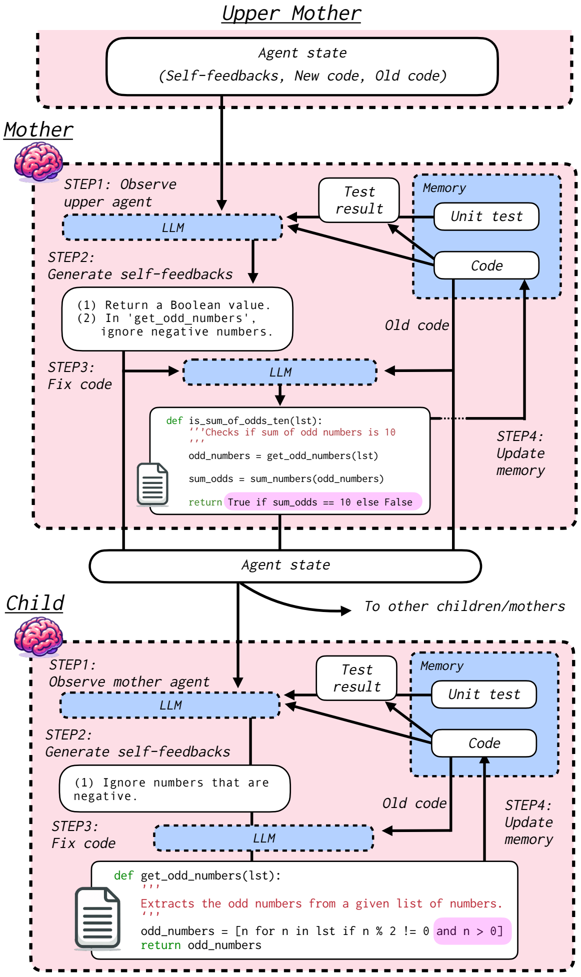
cation (Figure 3). It observes information from the upper Mother and uses it to modify the functions it is responsible for. The only difference is that the feedback it generates and the code before and after modification are used by lower-level agents (Child or Mother).

3.3 Self-organized Agent Process

The Self-organized Agent (SoA) framework is a distributed framework in which multiple agents (including Mother agents and Child agents) repeatedly generate and modify functions. The core of this framework lies in the principle of self-organization, where each agent functions independently without the need to directly observe the entire codebase. The hierarchical combination of Mother agents and Child agents forms an agent network that effectively constructs a single large-scale codebase. In this hierarchical structure, Mother agents decompose complex problems into more manageable smaller problems by dividing tasks and delegating them to the agents they have generated. Although each agent is independent, the agents as a whole can work efficiently towards the implementation of a single function. Despite the fact that the amount of code each agent generates, modifies, and manages is always small, the number of agents scales, allowing the amount of code generated to be increased indefinitely according to the difficulty of the problem. Detailed algorithms are presented in Algorithm 1 in the appendix.

Code Generation The code generation process in the SoA framework begins with the function's docstrings and unit tests. In the initial stage, there is only one initialized Mother agent, which is the root of the tree structure. Based on the input docstrings and unit tests, it generates docstrings and unit tests for subtasks and passes them to other agents it generates (see §3.2). If the tree structure reaches a predetermined depth, the tasks are passed to Child agents; otherwise, they are passed to newly generated Mother agents. By repeatedly proliferating and increasing the number of agents until the last agent, it is possible to generate large-scale code while keeping the amount of code managed by individual agents constant.

Code Modification Once code generation is complete, the process transitions to the code modification phase. First, the implementations of all agents are combined to create the final implementation. This final implementation is evaluated using the unit tests provided to the root Mother, and feedback is generated from the results. Since there are no agents higher than this root Mother, information from higher-level agents as shown in Figure 3 is not used. The modification process starts based on this feedback and propagates information from the root Mother agent to the Child agents. Each agent updates its implementation based on the received



(1) Return a Boolean value.
(2) In 'get_odd_numbers',
ignore negative numbers.

```
def is_sum_of_odds_ten(lst):
    '''Checks if sum of odd numbers is 10
    '''
    odd_numbers = get_odd_numbers(lst)

    sum_odds = sum_numbers(odd_numbers)

    return True if sum_odds == 10 else False
```

(1) Ignore numbers that are
negative.

```
def get_odd_numbers(lst):
    '''
    Extracts the odd numbers from a given list of numbers.
    '''
    odd_numbers = [n for n in lst if n % 2 != 0 and n > 0]
    return odd_numbers
```

Figure 3: Overview of code modification. Agents (Mother/Child) observe the state of Mother (feedback, old code, and updated code) and use this information to improve the functions for which they are responsible. The state of the upper agent is used to modify code by lower agents within the hierarchy. This state propagation promotes collaboration and information sharing throughout the hierarchy, enabling efficient code modification.

feedback, generates new feedback, and transmits it to lower-level agents (see § 3.2). Finally, the Child agents update their own implementations, and the process terminates (see § 3.1). This series of processes is repeated until a predetermined maximum number of iterations is reached.

4 Experiments

LLM Selection We used GPT3.5-turbo[3] for code generation and feedback generation.[4]

Baselines We compare SoA with several state-of-the-art code generation methods including AlphaCode (Li et al., 2022), Incoder (Fried et al., 2023), Codex (Chen et al., 2021), CoT (Wei et al., 2022), and Gemini Pro (Anil et al., 2023). Additionally, we evaluate the performance of various GPT-3.5-based agents, such as ChatGPT, Self-Edit (Zhang et al., 2023), and Reflexion (Shinn et al., 2023). These baselines are chosen to represent a diverse range of approaches, including single-agent and multi-agent systems, as well as those with and without self-debugging capabilities.

Agent Configuration To evaluate the effectiveness of the SoA framework, we selected the Reflexion agent as a baseline. Reflexion iteratively modifies code based on the given docstrings and automatically generated unit tests until it reaches the maximum number of iterations or passes the unit tests. The main difference between Reflexion and SoA is that Reflexion is composed of a single agent, while SoA is composed of self-organized multiple agents. In the SoA configuration, we set the maximum number of iterations for the learning loop to 8 and the maximum tree depth to 2. Additionally, following (Shinn et al., 2023), we provided a few-shot trajectory to the LLM.

Data and Tasks To evaluate the performance of automatic code generation, we used the HumanEval (Chen et al., 2021) benchmark. HumanEval is a set that includes diverse programming problems designed to measure the functional correctness of generated code. We used the Python language set for evaluation and followed the evaluation methodology of Reflexion (Shinn et al., 2023). In this process, multiple test cases are created for each generated code, and n test cases are randomly selected to construct a test suite. This test suite is used to verify whether the generated code functions correctly. We set 6 unit tests for Reflexion and 1 unit test for SoA.

4.1 Main Results

Table 1 compares the Pass@1 accuracy of the proposed method and the baseline. Comparing SoA with Reflexion, a strong baseline, SoA outperforms Reflexion by 5% in Pass@1. Considering that each agent in SoA does not see the entire code, this is a surprising result. This result suggests that self-organized agents can generate code that functions well as a whole without needing to oversee the

[3] gpt3.5-turbo-1106

[4] GPT-4 was not selected due to the high experimental cost required.

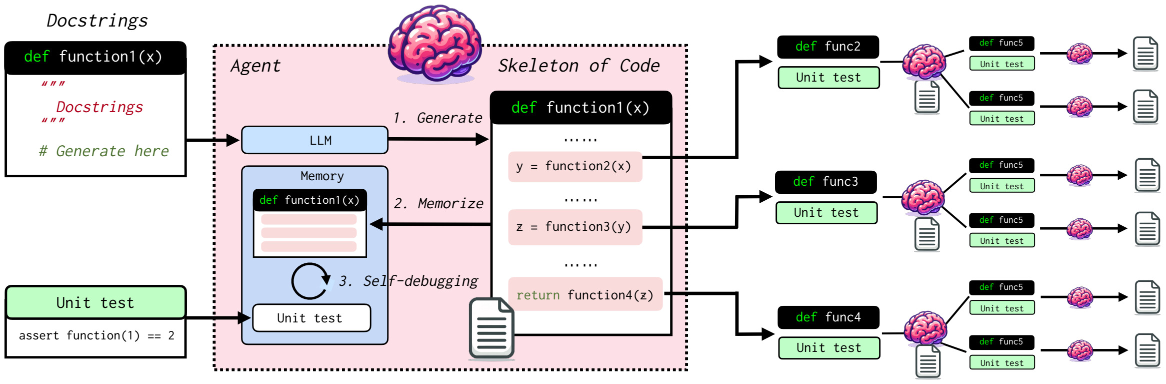


Figure 4: Overview of the SoA framework. Mother agents and Child agents hierarchically construct a network and perform function generation and modification. Mother agents delegate tasks to other Mother agents or Child agents, and each agent independently executes tasks while effectively implementing a single function as a whole.

Method		**SD**	**SO**	**Pass@1**
AlphaCode	(Li et al., 2022)	✘	✘	17.1
Incoder	(Fried et al., 2023)	✘	✘	15.2
Codex	(Chen et al., 2021)	✘	✘	47.0
Gemini Pro	(Anil et al., 2023)	✘	✘	67.7
GPT-3.5	CoT (Wei et al., 2022)	✘	✘	44.6
	ChatGPT	✘	✘	57.3
	Self-Edit (Zhang et al., 2023)	✔	✘	62.2
	Reflexion (Shinn et al., 2023)	✔	✘	66.5
	SoA (ours)	✔	✔	**71.4**

Table 1: Results of SoA and baselines on HumanEval. The score of ChatGPT is taken from Dong et al. (2023). **SD** indicates whether the agent uses self-debugging with unit tests, while **SO** denotes whether the agent employs self-organized multi-agent collaboration.

entire code, by independently implementing the functions assigned to them.

4.2 Analysis

One of the most critical aspects of our study is the efficiency of the self-organized multi-agent approach in large-scale code generation. To showcase the superior performance of SoA, we conducted a comprehensive comparative analysis between Reflexion, a state-of-the-art single-agent system, and our proposed multi-agent system. Using the HumanEval benchmark, we meticulously examined the overall scale of the code generated by both systems and the amount of code each agent independently generated and memorized. To ensure a fair comparison, we removed comments and docstrings from the HumanEval results and focused on the number of characters and tokens of pure code.

Figure 5 presents a visualization of the average amount of code generated by SoA and Reflexion from the perspective of individual functions and all functions. In the context of HumanEval, which requires the implementation of a single function, SoA's code amount is calculated by summing the code generated by each agent, while Reflexion's code amount is based on a single function. The *code amount per function* in SoA refers to the code generated by each individual agent, whereas in Reflexion, it is equivalent to the code amount of a single function. The results unequivocally demonstrate SoA's superiority over Reflexion in terms of the number of tokens per final code and the average number of characters per function. What is remarkable is that despite each agent in SoA handling significantly fewer tokens/characters compared to the single agent in Reflexion, the overall output generated by SoA is substantially greater. This finding underscores the exceptional scalability of SoA, indicating its ability to handle increasingly complex tasks by seamlessly adding more agents to the system. Our results suggest that by increasing the depth of the agent hierarchy and introducing more Mother agents, SoA can generate even larger-scale code by efficiently distributing the workload among multiple agents. As the tree structure becomes deeper, the system exhibits an infinite scaling potential, enabling the generation of increasingly complex and extensive codebases while ensuring that each agent handles a manageable portion of the code. Each agent can maintain a manageable amount of code while theoretically allowing for an indefinite increase in the overall code generation capacity.

This distributed approach empowers SoA to significantly scale up its ability to tackle large-scale and complex coding tasks with remarkable efficiency and high quality, far surpassing the limi-

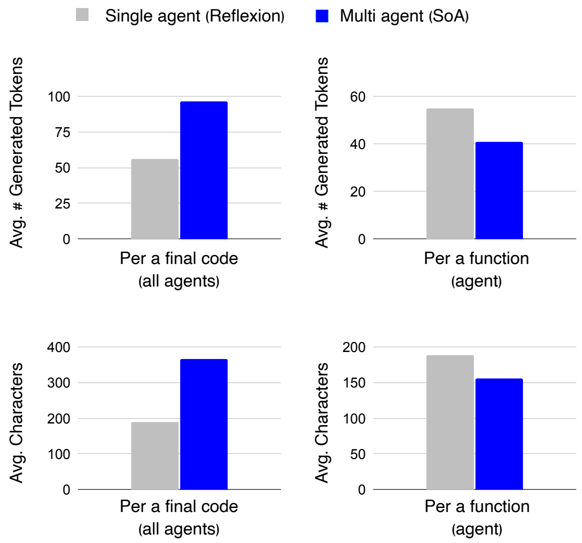


Figure 5: Comparison of code generation amount between SoA (mulit-agent) and Reflexion (single agent).

tations encountered by single-agent systems like Reflexion, where a sole agent is responsible for managing and generating the entire codebase.

5 Related Work

LLM Agents Recent advancements in LLM agents, such as ReAct (Yao et al., 2023b), Reflexion (Shinn et al., 2023), Toolformer (Schick et al., 2023), and Self-Refine (Madaan et al., 2023), have primarily focused on single-agent approaches, where one agent is responsible for both generation and modification tasks. Among these, Reflexion (Shinn et al., 2023) has gained significant attention in the field of code generation due to its outstanding performance. However, despite their strengths, these single-agent approaches face inherent limitations when it comes to generating and modifying large-scale codebases. To address these limitations and push the boundaries of what is possible with LLM agents, we propose SoA, a novel multi-agent framework that harnesses the power of self-organization and collaboration. While we intentionally adopted simple agents for SoA in this work, our framework is flexible enough to incorporate more sophisticated and powerful methods (Zhong et al., 2024; Zhou et al., 2023) and other state-of-the-art LLMs [5], further enhancing its potential for large-scale code generation and modification.

Multi-Agent Collaboration for Software Development In recent years, several multi-agent-based approaches have emerged as promising solutions for software development, such as MetaGPT (Hong et al., 2023), ChatDev (Qian et al., 2023), Self-collaboration (Dong et al., 2023), and AgentCoder (Huang et al., 2023). These methods typically personify agents and assign them specific names or occupational roles, such as programmers, project managers, or QA engineers, to allocate tasks. While this approach has shown promise, our method takes a different and more flexible approach. Instead of assigning fixed occupational roles, we subdivide agent capabilities based on *code functionality*, allowing each agent to demonstrate its expertise without being constrained by predefined roles. This fine-grained task allocation enables more flexible problem-solving and adaptation to the complexity of the software development process. Moreover, by incorporating the concepts of self-organization and self-proliferation, our agents can dynamically scale up the overall code volume based on the difficulty of the problem at hand, providing a highly adaptable and efficient framework for large-scale code generation and modification.

Macro vs. Micro Perspectives While both multi-agent-based methods (Hong et al., 2023; Qian et al., 2023; Dong et al., 2023; Huang et al., 2023) and our proposed SoA framework share the common goal of automating software development, they address different technical aspects of the process. Existing multi-agent methods primarily focus on optimizing the macro structure of software development, such as project management and task allocation. In contrast, our method takes a more micro-level perspective, focusing on the elemental technologies of code generation and modification. These approaches are not mutually exclusive but rather complementary, offering a more comprehensive solution to the challenges faced in automatic software development. By combining the strengths of both macro and micro-level approaches, we can create a powerful and holistic framework that efficiently handles the complexities of large-scale code generation and modification.

Prompt Engineering Tree-of-Thought (ToT) (Yao et al., 2023a) and Skeleton of Thought (SoT) (Ning et al., 2023) are prompt engineering techniques that utilize tree-like structures. ToT represents reasoning steps as nodes to explore correct reasoning paths, while SoT

[5] https://claude.ai/

generates a skeleton of the answer and completes the contents in parallel to decrease generation latency. In contrast, SoA uses a tree structure with agents as nodes, focusing on their collaboration and self-organization to generate and modify code efficiently.

6 Conclusion

In this work, we introduced Self-organized Agents (SoA), a novel multi-agent framework for efficient and scalable automatic code generation and optimization using large language models (LLMs). SoA addresses the limitations of single-agent approaches in handling large-scale, complex codebases by leveraging the power of self-organization and distributed code generation. In SoA, self-organized agents operate independently to generate and modify code components while seamlessly collaborating to construct the overall codebase. A key feature of our framework is the automatic multiplication of agents based on problem complexity, allowing for dynamic scalability and enabling the overall code volume to be increased indefinitely according to the number of agents, while the amount of code managed by each agent remains constant.

We evaluated SoA on the HumanEval benchmark and demonstrated its superior performance compared to Reflexion, a state-of-the-art single-agent system, with SoA achieving a 5% improvement in terms of Pass@1 accuracy. Furthermore, our in-depth analysis revealed SoA's remarkable scalability, as each agent in SoA handles significantly less code compared to the single-agent baseline, yet the overall generated code is substantially greater. These results highlight the effectiveness of SoA in generating and optimizing large-scale code efficiently and with high quality.

However, it is essential to acknowledge the limitations of the current implementation of SoA. The framework's performance may be affected by the choice of LLM and the quality of the generated unit tests. Additionally, SoA has been evaluated on a limited set of programming tasks, and its effectiveness in handling more complex, real-world software development projects remains to be investigated. Furthermore, the communication and collaboration mechanisms among agents in SoA can be further optimized to improve efficiency and fault tolerance.

Despite these limitations, we believe that the SoA framework has significant potential for future research and development in the field of automatic software development.

References

Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Slav Petrov, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, Timothy P. Lillicrap, Angeliki Lazaridou, Orhan Firat, James Molloy, Michael Isard, Paul Ronald Barham, Tom Hennigan, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, Ryan Doherty, Eli Collins, Clemens Meyer, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, George Tucker, Enrique Piqueras, Maxim Krikun, Iain Barr, Nikolay Savinov, Ivo Danihelka, Becca Roelofs, Anaïs White, Anders Andreassen, Tamara von Glehn, Lakshman Yagati, Mehran Kazemi, Lucas Gonzalez, Misha Khalman, Jakub Sygnowski, and et al. 2023. Gemini: A family of highly capable multimodal models. *CoRR*, abs/2312.11805.

W Ross Ashby. 1947. Principles of the self-organizing dynamic system. *The Journal of general psychology*, 37(2):125–128.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison 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, Joshua 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. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023. Self-collaboration code generation via chatgpt. *CoRR*, abs/2304.07590.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, and Chenglin Wu. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *CoRR*, abs/2308.00352.

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

Mosh Levy, Alon Jacoby, and Yoav Goldberg. 2024. Same task, more tokens: the impact of input length on the reasoning performance of large language models. *CoRR*, abs/2402.14848.

Jiaqi Li, Mengmeng Wang, Zilong Zheng, and Muhan Zhang. 2023a. Loogle: Can long-context language models understand long contexts? *CoRR*, abs/2311.04939.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023b. Starcoder: may the source be with you! *CoRR*, abs/2305.06161.

Yujia Li, David H. 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. 2022. Competition-level code generation with alphacode. *CoRR*, abs/2203.07814.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. *CoRR*, abs/2308.07124.

Xuefei Ning, Zinan Lin, Zixuan Zhou, Huazhong Yang, and Yu Wang. 2023. Skeleton-of-thought: Large language models can do parallel decoding. *CoRR*, abs/2307.15337.

OpenAI. 2023. GPT-4 technical report. *CoRR*, abs/2303.08774.

Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. 2023. Communicative agents for software development. *CoRR*, abs/2307.07924.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Uri Shaham, Maor Ivgi, Avia Efrat, Jonathan Berant, and Omer Levy. 2023. Zeroscrolls: A zero-shot benchmark for long text understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 7977–7989. Association for Computational Linguistics.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023a. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023b. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 769–787. Association for Computational Linguistics.

Lily Zhong, Zilong Wang, and Jingbo Shang. 2024. LDB: A large language model debugger via verifying runtime execution step-by-step. *CoRR*, abs/2402.16906.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. *CoRR*, abs/2310.04406.

A Pseudocode

Algorithm 1 Generate Code with Self-organized Agent Framework

```
Input: $docstrings$: Docstrings for the function, $unit\_tests$: List of unit tests, $max\_depth$: Maximum depth of the agent
    hierarchy, $max\_iterations$: Maximum number of code modification iterations
Output: The final generated code

Initialize the root Mother agent with $docstrings$ and $unit\_tests$

function GENERATEAGENT($agent$, $depth$, $subtask\_docstrings$, $subtask\_unit\_tests$)
    if $depth - 1$ = $max\_depth$ then
        $next\_agent \leftarrow$ new ChildAgent
    else
        $next\_agent \leftarrow$ new MotherAgent
    end if
    Assign $subtask\_docstrings$ and $unit\_tests$ to $next\_agent$
    GENERATE($next\_agent$, $depth + 1$)
end function

function GENERATE($agent$, $depth$)
    if $depth$ = $1$ then                                                        ▷ Root Mother
        $skeleton \leftarrow$ Generate skeleton from $agent.docstrings$ and $agent.unit_tests$
        $agent.code \leftarrow skeleton$
        for each $subtask\_docstrings$, $subtask\_unit\_tests$ in subtasks do
            GENERATEAGENT($agent$, $depth$, $subtask\_docstrings$, $subtask\_unit\_tests$)
        end for
    else if $depth$ = $max\_depth$ then                                          ▷ Child
        Generate code for $agent.subtask\_docstrings$ and $agent.subtask\_unit\_tests$
        $agent.code \leftarrow$ generated code
    else                                                                         ▷ Mother
        Generate code for $agent.subtask\_docstrings$ and $agent.subtask\_unit\_tests$
        $agent.code \leftarrow$ generated code
        for each $subtask\_docstrings$, $subtask\_unit\_tests$ in subtasks do
            GENERATEAGENT($agent$, $depth$, $subtask\_docstrings$, $subtask\_unit\_tests$)
        end for
    end if
end function

function MODIFY($agent$, $test\_result$, $upper\_agent\_observation$)
    Generate feedback for $agent$ based on $test\_result$ and $upper\_agent\_observation$
    Update $agent$'s code based on feedback
    for each $subagent$ in $agent.subagents$ do
        Evaluate $subagent.code$ using $subagent.unit\_tests$ to get $subagent\_test\_result$
        MODIFY($subagent$, $subagent\_test\_result$, feedback and code changes)
    end for
end function

Start code generation with GENERATE($root\_mother$, 1)

for each iteration in $max\_iterations$ do
    Combine implementations from all agents to create $final\_implementation$
    Evaluate $final\_implementation$ using $unit\_tests$ to get $test\_result$
    Modify the code starting from $root\_mother$ with MODIFY($root\_mother$, $test\_result$, None)
end for

return The final implementation combined from all agents
```