ALICE: Aligning Language models with Interactive Code Execution in game engines

Yang Su, Young He, Joy Wang, Cheng Fei
ys724, jh2795, zw673, cf482@cornell.edu

Abstract

Alignment of Large Language Models' (LLMs) in code generation is challenging, especially in environments that require precise control, i.e., robotics simulation and physics engines. Code generated by LLMs is hard to execute and deploy as current autonomous agent frameworks primarily focus on the success of instruction execution while ignoring operational costs and interactive user guidance. For example, current LLM agents often complete tasks like "draw a building" without seeking user specifications about the details in the building's size, location, or shape. We are hence driven to develop an efficient **meta-agent collaboration system** by integrating interactive code generation and parallelizable system execution feedback. This framework called ALICE, evaluated on a terrain environment in game engines, has shown to be more efficient and preferable by users. ALICE is also orthogonal to popular agent frameworks like ReAct, ToT, Reflexion due to its modular nature, while combing their advantages in different aspects of human-AI and AI-AI interaction.

1 Background

1.1 Code generation with LLMs

Code generation with Large Language Models (LLMs) has been widely adapted by the following two category of design.

- Copilot: this design allows users to input documentation comments and convert them into code. Users can modify, run, and debug the code as they would in typical software development processes, making it a highly **controllable** system.
- Autonomous agents: this approach focuses on generating entire code scripts to fulfill user instructions. Typically, models within this system interact and utilize tools to search, execute, and debug the code they generate, continuing until a solution is achieved or a maximum trial limit is reached. This method is considerably **uncontrollable** due to minimal interaction between the user and the agent system.

Bridging the functionalities of Copilot and Autonomous Agents, ALICE enables coding in natural language and is specifically engineered to enhance efficiency in following instructions within interactive environments. Unlike autonomous agents, which operate independently of human input and learn from experiences to determine optimal actions, ALICE actively identifies potential issues during execution and solicits clarifying questions. Through iterative interactions, ALICE delivers results that are more precise and better aligned with user requirements.

1.2 Alignment

Fine-tuning LLMs is important for optimizing ALICE for complex and interactive environments. While ALICE supports various advanced fine-tuning methods, including Reinforcement Learning from Human Feedback (RLHF), Decision Process Optimization

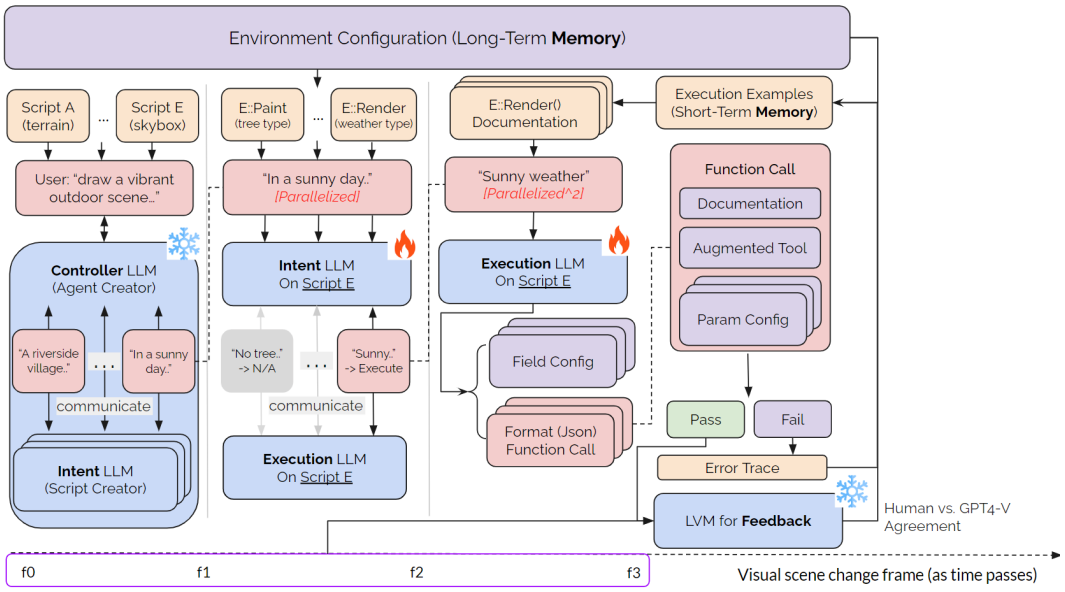

Figure 1: An Overview of the ALICE Workflow. Fine-tunable models are labeled with the fire symbols, and frozen models are labeled with the snow symbols. For components with different colors, purple refers to structured arguments or configurations in the game engine, orange refers to textual prompts or examples, blue refers to different large language (vision) models, red refers to textual data generated from the model or provided by users, and green refers to that the program successfully executes.

(DPO), and Reward Augmented Training (RAFT) which leverage human feedback to dynamically refine the model's responses for improved alignment with human expectations and operational efficiency, this research opts for Supervised Fine-Tuning (SFT) due to its simplicity and effectiveness. SFT involves training ALICE on a carefully curated dataset specifically designed to simulate the interaction dynamics typical of interactive settings in game engines.

1.3 Multi-Agent architecture

The Multi-Agent architecture allows ALICE to utilize various agents specialized in different aspects of the interaction and execution processes within interactive settings such as game engines. Each agent is designed to handle specific tasks, ranging from communicating with users and other agents, modifying agents, creating higher-level tools, generating code, to executing commands and providing feedback. This division of labor increases the efficiency of the system by processing parallelized environment changes and improves the precision and user instruction following ability of ALICE.

2 Method

We focus on code execution with LLMs in game engines like Unity, leveraging the fully controllable, observable, and modularized virtual environment they provide. We propose an efficient *meta-agent collaboration* framework named ALICE, designed to follow user instructions through active communication, aligning with their intentions, and generating code based on continuous feedback and interaction. This section will discuss 3 main components of ALICE, Controller LLM, Intent LLM and Execution LLM.

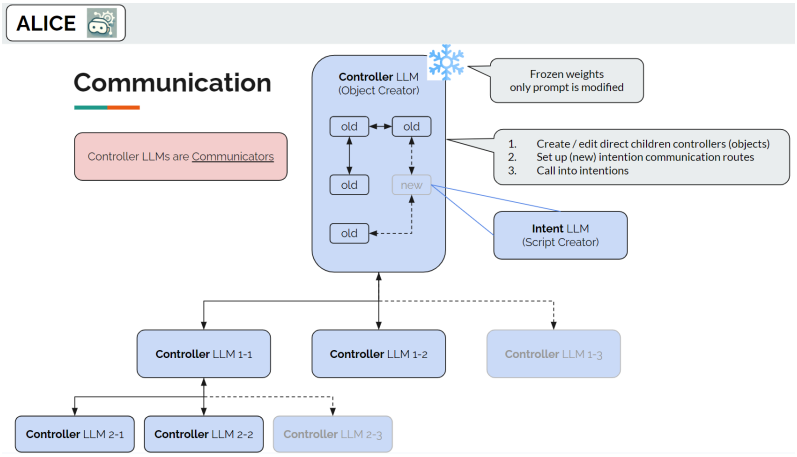

Figure 2: An example of a Controller LLM.

2.1 Controller LLM for Communication

We start by delineating what we mean by constructing a *meta-agent collaboration*.

Existing multi-agent collaboration frameworks like AutoGenWu et al. (2023), MetaGPT Hong et al. (2023a) and AutoGPT operate under the premise that each agent or LLM is pre-defined to be equipped with specific roles, or instruction prompts that dictate their responses to given instructions. These roles often include tool-use capabilities, and the agents are designed by humans to communicate in specific ways. This design approach relies heavily on carefully crafted prompts and does not adapt well to different environments. Conversely, frameworks tailored for specific settings like Stanford Town Park et al. (2023a) or GPT-engineer are overly specialized and fail to generalize across various domains.

We propose a novel *meta-agent collaboration* framework as the follows. Specifically, we setup a controller LLM with the following fixed *operations* that manage and interact with a suite of intention LLMs, which can be treated as general LLM agents seen in other frameworks.

- Create/Edit/Delete: modifying another controller's instruction prompt, creating or deleting a new controller by writing done its instruction prompt.
- Route Setup: setup communication route for a new intention LLM to interact with other intention LLMs it controls.
- Call: send instruction to one intention LLM under its control.

This configuration enables the controller LLM to act as the creator of other LLMs, which are essentially LLMs with different instruction prompt. Note that the LLMs it create are also controllers, i.e. they are equipped with the same 3 operations as their parent controller. The controller LLMs do not do any task, instead, they are communicators with other LLMs and the user. An example of a controller LLM is in Figure 2.

2.2 Intent LLM for Alignment

Intent LLMs are named by their nature of following personalized instructions. Each intent LLM is in charge of all the changes of a special code script, that is called a *game component*, and a execution LLM.

Game Components Physic engines usually use a component-based architecture, where each object in the virtual scene can have various components attached to them that perform operations in parallel. This modular approach allows developers to compose behavior and properties by attaching different components to objects, such as renderers, scripts, colliders, or custom components. A game component is a script that an object can have, for example, a

Terrain.cs script defines how an area of geographic structure is rendered, containing methods (tools) to create trees, grass and mountains. It can be treated as a class of any API library. As such, we attach a fine-tunable LLM to each API script we care about, by reading its class documentation in its declaration, attribute getter and setter, to each method (function) example usage as part of the model prompt.

The intent LLM is equipped with the following operations.

- Create/Edit/Delete: modifying tools from the API it is in charge of, for example, it can add or delete a RAG or add a vision feedback system for the Terrain.cs API when it see fits to reduce memory overhead or boost execution LLM's performance.
- Route Setup: create inherited scripts from existing APIs, which creates new intent LLM, upon doing this, it must ask the controller LLM for the communication route of the new intention.
- Call: call into the execution LLM with the task it should do.

This setup allows the intent LLM to be a creator for code scripts, especially higher-level codes from existing codes. We can analogy this design to the idea of *skill library* in Voyager (insert link), where the agent is prompted to adapt new tools that incorporates existing atomic tools. Each intent LLM that are attached to the existing base APIs are fine-tunable, and they can be tuned to be a more personalized version by adapting changes in the inherited API class. In production, there could be potentially thousands of API scripts, whereas only a minimal subset of those need to be tuned, for example, for people who only care about lightning changes in a virtual architectural design environment, only the scripts that are related to shaders need to be inherited. Hence, the intent LLMs can be better aligned to follow personalized instructions than generally aligning to all sorts of user requirements. An example of a intent LLM is in Figure 3.

2.3 Interactive Code Execution LLM

An execution LLM controls how an API script should be used when receiving user instructions related to its component update, augmented with tools (functions) written in the given script. In practice, we give it the header file (Terrain.h).

Different from a general instruct-tuned code generation model, an interactive code execution LLM generates code via multi-turn conversation with the game engine and user feedback. In short, as we modularize the API separation, each execution model can criticize the given instruction for its game component by asking back the parent intent LLM for clarification in code execution with unknown parameters. For example, to add trees in a Terrain component, the user might want to further instructs their tree density and range requirements before letting the model to proceed the code generation task. We collect feedback from each execution LLM send them to the upper level to wait for the user or the controller with a broader knowledge about all the APIs to respond. This is similar to ToT (Tree of Thought) but optimized for *ta-agent collaboration*in our scenario.

The interactive code execution LLM is equipped with the operations to generate code, incorporating feedback from the tools it is given from the intent LLM, or report the final code it writes that is ready for execution.

2.4 Agent Strategies

A big advantage of the ALICE system is that it is orthogonal to popular agent strategies by its nature of design. The operations that are primarily equipped to each LLM component can be boosted by planning ahead, reasoning through, self-critic-ed, and virtually executed as actions to better follow user instructions.

To use ReAct, for example, the intent LLM can be equipped with the RAG and communication tool with user to better fulfill user instruction.

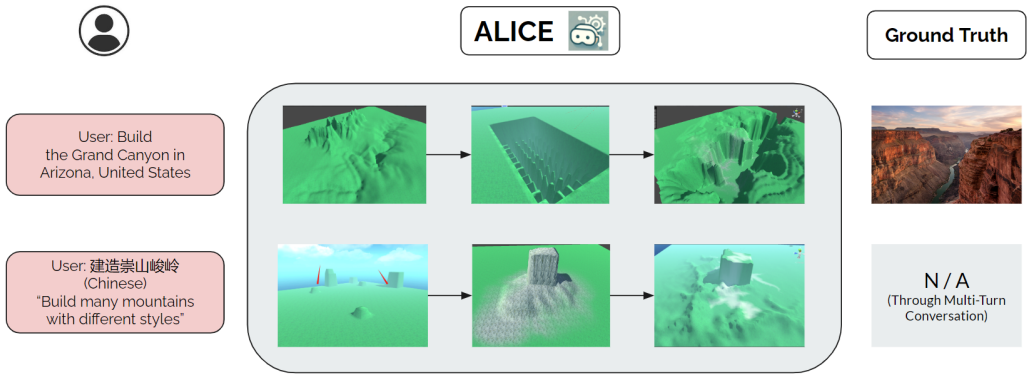

Figure 3: ALICE example input and output. The *ALICE (icon)* result from left to right shows the result of GPT-3.5 before, as fine-tuning progress, and after alignment. The ground truth is not included in the training.

3 Experimental analysis

3.1 Setup

We conducted our experiments with 20 participants from diverse backgrounds. Each participant was asked to give five instructions and to rate the model-generated results on a score from 1 to 5. A score of 1 indicates that the output is completely irrelevant or fails to execute the instruction, while a score of 5 signifies that the output perfectly aligns with the instructions. The higher the score, the better the quality of the output. To assess the effectiveness of different models and ALICE components, we required participants to provide the same instructions based on different models and ALICE components.

3.2 Model and configuration

The models we used included GPT-3.5-Turbo as a baseline, GPT-4-Turbo, Llama-3-8B, and Llama-3-8B fine-tuned. For the fine-tuned Llama-3-8B, we collected 1,200 correct pairs of instructions and code as training data, where each code sample was the optimal code from the final round of a multi-turn dialogue. We exclusively used GPT-4-Turbo for generating training data to ensure optimal performance. Additionally, we tested four different ALICE components: the baseline configuration with only an execution component, the execution components with feedback, an Intention-Execution configuration that includes both intention and execution components, and the complete ALICE configuration with all three components.

3.3 Evaluation

To better evaluate our model, we employed three metrics: Execution Score, Satisfaction Score, and Inference Speed. These metrics are designed to assess the quality and efficiency of the code generated by the model. The Execution Score is the average rating given by users or GPT-4 to the model-generated C# code. The Satisfaction Score is the average rating given by users or GPT-4 to the model's output. Inference Speed measures the time it takes for the model to execute, while human interaction time is excluded. For parallelizability, we start at most 4 threads to virtually run each local model in batch, or for API-based models, we simply collect all inputs from the same-level LLMs (intention and execution LLMs) and get their results in batch back to the lower-level LLMs.

Given the vague and unpredictable nature of human instructions, we employed an oracle AI agent (GPT- 4) designed to mimic human interactions across various social statuses. This

approach categorizes user groups by the precision level of their control over the instructions. We generated datasets that reflect a range of instruction difficulties and systematically assess the performance of multi-turn interactions.

3.4 Results

Model	**Execution Score (human/GPT-4V)**	**Satisfaction Score (human/GPT-4V)**
GPT-3.5-Turbo (with ReAct)	4.52 / 4.48	2.78 / 2.85
GPT-4-Turbo (with ReAct)	4.74 / 4.66	3.07 / 3.16
Llama-3 (with ReAct)	2.56 / 2.36	2.11 / 1.92
Llama-3 fine-tuned (with ReAct)	4.07 / 3.84	3.55 / 3.46

Table 1: Comparison of the code and the ouputs generated by different models

Config	**Execution Score (human/GPT-4V)**	**Interaction Score (human/GPT-4V)**	**Inference Speed (parallelized/un-parallelized)**
Execution Only	2.11 / 1.89	0.23 / 0.08	1.2s / 1.2s
Execution Feedback	2.93 / 2.35	1.53 / 1.13	8.7s / 8.7s
Intent-Execution	3.20 / 3.17	2.61 / 2.38	16.9s / 88.3s
Control-Intent-Execution (ALICE)	**4.07 / 3.84**	**3.55 / 3.46**	174.0s / 2164.5s

Table 2: Performance comparison of different ALICE components. All experiments in this section were conducted using fine-tuned Llama3-8B.

4 Discussion

4.1 Advantages

ALICE distinguishes from traditional multi-agent collaboration framework by its allowance of dynamic creation and management of agents and tools.

- Instruction Prompts as Agent Creation: by employing a modular approach where different agents are created to handle distinct parts of the task sequence, ALICE achieves a level of personalization and responsiveness that aligns closely with user intentions.
- Tools as Agent Instruction: In ALICE, tools are not merely supplementary; they are integral to how agents perceive and execute tasks. Tools can be used to create other tools that affect how each agent behave. Agent creation via prompt is also a core tool of the controller LLM.

4.2 Limitations

One major challenge of ALICE is the high costs associated with its evaluation, particularly in interactive setting where we need to evaluate the agreement between AI-AI and human-AI interaction. Secondly, its effectiveness has been primarily tested within game engines, raising questions about its adaptability to other environments that lack similar structure and contractility. Its reliance on game components may restrict its utility in real-world environments or less structured scenarios, where such elements are not as clearly defined, manipulable or observable.

5 Related Works

Agent strategies. A wide range of works have examined how prompting and other techniques can improve agents' behavior. MCPlanner (Wang et al. 2023b) utilizes Minecraft as a testbed for developing planning algorithms that handle complex, dynamic environments. Similarly, ReAct (Yao et al. 2023b) focuses on reactive planning under uncertainty, enhancing an agent's ability to adapt to unforeseen changes in its environment. Tree of Thoughts (Yao et al. 2023a) and the more classical Chain of Thoughts (Wei et al. 2023) are prompting techniques that significantly increase the performance of generative agents. Toolformer (Schick et al. 2023) introduces an innovative approach by integrating tool use capabilities into transformer-based models, enabling agents to manipulate and interact with their environment in more sophisticated ways. Language Agent Tree Search (Zhou et al. 2023) further extends this by combining natural language processing with decision-making processes, providing a framework where agents can plan their actions based on linguistic input. ALICE is similar to these work in that it can be considered as an extension to some of these agent strategies (e.g., ReAct), but it is more capable as it integrates directly to the game engine for actions.

Agents in interactive environments. There have been a variety of works demonstrating how LLM agents can be used in interactive environment to complete a variety of tasks. MineDojo (Fan et al. 2022), as well as Voyager (Wang et al. 2023a), explore the usage of generative agents for world exploration in Minecraft. Stanford Town (Park et al. 2023b), on the other hand, explored using many agents and having them interact with each other in a closed simulacra. Aside from simulated environments, there are also works like SayCan (Ahn et al. 2022) that experimented with translating natural language instructions to actions of a robotics arm in a real world environment. ALICE is similar to these works in that it interacts with a simulated environment, but differs in that it directly interacts with the game engine to perform tasks.

Language models for code generation. Aside from their stellar performance in natural language tasks, LLMs have also shown promising results for code generation. CodeT (Chen et al. 2022) leverages advanced transformer architectures to generate high-quality code snippets, significantly easing the programming process for developers by automating routine coding tasks. Similarly, Code LLaMa (Rozière et al. 2024) extends the utility of large language models to the coding domain, enabling them to understand and generate code in multiple programming languages with high accuracy. Code as Policy (Liang et al. 2023) introduces a novel approach by using code generation as a mechanism for policy enforcement in software environments, thereby enhancing security and compliance. ALICE is similar to these works in that it generates code for the Unity engine to execute, but it is also different in that its direct interaction with the game engine, as well as its controller-intention-execution framework, allows for much more granulated control of its code output.

Multi-agent collaboration. The concept of using multiple agents to collaboratively solve tasks is also well-studied. Stanford Town (Park et al. 2023b), as discussed above, utilized the collaborative nature of several generative agents to observe social tendencies. On the other hand, AutoGPT (Yang et al. 2023) emphasizes autonomous operations, enabling agents to perform a variety of tasks without human intervention, which is crucial for scalable agent deployment in diverse scenarios. MetaGPT (Hong et al. 2023b) extends these concepts by focusing on meta-programming capabilities, allowing agents to adapt and optimize their behavior dynamically in response to environmental changes and task requirements. AgentVerse (Chen et al. 2023) takes a holistic approach by creating a versatile ecosystem where multiple agents can interact, learn, and collaborate, pushing the boundaries of what is achievable in cooperative multi-agent systems. CoELA (Zhang et al. 2024) integrates cognitive elements into agent designs, enhancing their ability to understand and react to complex multi-agent dynamics. ALICE also deploys various agents (controller, intention, execution, etc.), but they are organized in a particular structure that is suitable for game engine code execution.

6 Conclusion

In conclusion, the ALICE framework combines multiple specialized LLMs within game engines to efficiently follow complex user instructions. By integrating Controller, Intent, and Execution LLMs, ALICE excels in managing communications, aligning user intents with game components, and generating task-specific code. This precise control and dynamic interaction overcome limitations of traditional instruction following models in virtual environments through enhanced human-AI interaction.

ALICE can be especially useful in data generation with active human intervention. In data-limited environments such as virtual reality, robotics simulation, and autonomous driving, the ability to generate high-quality, actionable data from sparse inputs is critical. This positions ALICE as an efficient and adaptive AI solution in a variety of advanced simulation technology applications.

References

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil J Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. Do as i can, not as i say: Grounding language in robotic affordances, 2022.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests, 2022.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors, 2023.

Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. Minedojo: Building open-ended embodied agents with internet-scale knowledge, 2022.

Sirui Hong, Xiawu 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*, 2023a.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework, 2023b.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control, 2023.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior, 2023a.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior, 2023b.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models, 2023a.

Zihao Wang, Shaofei Cai, Guanzhou Chen, Anji Liu, Xiaojian Ma, and Yitao Liang. Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents, 2023b.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

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.

Hui Yang, Sifu Yue, and Yunzhong He. Auto-gpt for online decision making: Benchmarks and additional opinions, 2023.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023b.

Hongxin Zhang, Weihua Du, Jiaming Shan, Qinhong Zhou, Yilun Du, Joshua B. Tenenbaum, Tianmin Shu, and Chuang Gan. Building cooperative embodied agents modularly with large language models, 2024.

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