AgentScope: A Flexible yet Robust Multi-Agent Platform

Dawei Gao[†], Zitao Li[†], Weirui Kuang[*], Xuchen Pan[*], Daoyuan Chen[*], Zhijian Ma[*], Bingchen Qian[*], Liuyi Yao, Lin Zhu, Chen Cheng, Hongzhu Shi, Yaliang Li[‡], Bolin Ding[‡], Jingren Zhou

Alibaba Group

Abstract

With the rapid advancement of Large Language Models (LLMs), significant progress has been made in multi-agent applications. However, the complexities in coordinating agents' cooperation and LLMs' erratic performance pose notable challenges in developing robust and efficient multi-agent applications. To tackle these challenges, we propose **AgentScope**, a developer-centric multi-agent platform with message exchange as its core communication mechanism. Together with abundant syntactic tools, built-in resources, and user-friendly interactions, our communication mechanism significantly reduces the barriers to both development and understanding. Towards robust and flexible multi-agent application, AgentScope provides both built-in and customizable fault tolerance mechanisms while it is also armed with system-level supports for multi-modal data generation, storage and transmission. Additionally, we design an actor-based distribution framework, enabling easy conversion between local and distributed deployments and automatic parallel optimization without extra effort. With these features, AgentScope empowers developers to build applications that fully realize the potential of intelligent agents. We have released AgentScope at `https://github.com/modelscope/agentscope`, and hope AgentScope invites wider participation and innovation in this fast-moving field.

1 Introduction

Multi-agent systems, differing from single-agent systems, require collaborative efforts from multiple agents working in concert (Wang et al., 2023; Xi et al., 2023). With the advancement of Large Language Models (LLMs) (Ouyang et al., 2022; OpenAI, 2023; Touvron et al., 2023a,b), multi-agent applications have made great process in both research and industrial communities, including software engineering (Hong et al., 2023), society simulation (Park et al., 2023), intelligent assistant (Wu et al., 2023; AutoGPT-Team, 2023). Although significant progress has been made in multi-agent scenarios, there are still major challenges remaining in multi-agent application development.

Developing a multi-agent application is significantly more complex. Unlike single-agent setups where an agent solely interacts with users, the development in the multi-agent scenario requires careful creation and management of multiple models and agents (Wang et al., 2023; Xi et al., 2023). That poses high requirements on both versatility and handiness for a platform. In particular, agents in a multi-agent application can specialize at different functions via different initial configurations; a multi-agent application may require agents to be executed in a standardized operating procedure (SOP) or a more dynamic workflow; the communication pattern between agents can be varying from one-to-one or broadcasting (e.g., a discussion group of agents). Thus, from the developers' point of view, they would expect a handy platform that can provide concise and clear programming patterns when taking care of all these aspects, accelerating and facilitating the development cycle. However, achieving versatility and handiness simultaneously requires careful design and taking trade-offs.

Aberrations are tinderboxs in a multi-agent system. Although large language models have advanced rapidly, they still struggle with issues like hallucination (Rawte et al., 2023; Zhang et al., 2023b) and inadequate

[†]Co-first authors.
[*]Equal contribution.
[‡]Corresponding authors, email address: {yaliang.li, bolin.ding}@alibaba-inc.com

instruction-following (Fu et al., 2019; Zhang et al., 2023a). Besides, an agent can be equipped with various tools, but those tools introduce additional uncertainties (e.g., accessibility to a database or the internet). From the perspective of multi-agent system robustness, a single unexpected error or response can propagate to the whole system, causing a series of cascading effects if not handled properly. Thus, it is crucial for multi-agent applications to autonomously detect and handle incorrect responses from LLMs. While LLMs may assist in identifying and managing these errors, it remains a challenge to determine whether they can resolve errors on their own and to automatically provide the necessary information for error correction. Consequently, designing fault-tolerant that incorporate LLMs is a key challenge in the development of multi-agent applications.

Compatible multi-modal data is highly systematic. There is increasing number of agent or LLM applications targeting on multi-modal content generation. Supporting multi-modal data (Su et al., 2023; Betker et al., 2023) in multi-agent applications requires a comprehensive and systematic approach. This includes considerations for data storage, presentation, user interaction, message transmission, communication, etc. However, meeting these requirements presents new challenges, includes ensuring data consistency across different formats, maintaining high performance during data transmission and agent communication, and avoiding the introduction of complex concepts for both developers and users. Although there are application-specific solutions, there are no general platform-level programming interfaces to support multi-modal applications.

Distributed applications bring extra programming difficulties and system design challenges. An industrial-oriented scenario for multi-agent applications is that the agents are owned by different organizations and run on different machines because the agents are equipped with unique private knowledge or patented tools. To support such applications, it usually requires the application developers to have professional knowledge on distributed system programming and optimization in the design phase. Besides, distributed applications usually require great extra effort in the development and testing, especially when debugging and diagnosing issues spread across distributed processes or agents. Moreover, integrating advanced features like multi-modal data processing pose additional challenges in a distributed setting, when the agents requires different time to accomplish the sub-tasks or the generated contents have are very heterogeneous. Poor design decisions can result in excessive communication overhead between agents. Therefore, it's challenging for developers to address these issues effectively to ensure effective system operations.

To address these challenges, we introduce **AgentScope**, a novel multi-agent platform designed for developers with varying levels of expertise. AgentScope is well designed with a message exchange communication mechanism that embodies great usability, robustness and efficiency. We underscore the salient features of AgentScope as follows:

Exceptional Usability for Developers: AgentScope is designed with a fundamental emphasis on ease of use, particularly for developers with varying levels of expertise. By implementing a procedure-oriented message exchange mechanism, AgentScope ensures a smooth learning curve. To further alleviate the programming burdens, AgentScope offers an extensive suite of syntactic utilities, including various pipelines and an information sharing mechanism. Coupled with rich built-in resources and integrated user interaction modules, AgentScope makes programming a multi-agent application much more enjoyable than ever.

Robust Fault Tolerance for Diverse LLMs and APIs: As the scale and scope of models and APIs expand, a robust fault-tolerance mechanism in multi-agent application becomes paramount. AgentScope integrates a comprehensive service-level retry mechanism to maintain API reliability. AgentScope equips with a set of rule-based correction tools to handle some obvious formatting problem in the responses of LLMs. Moreover, AgentScope offers customizable fault tolerance configurations, enabling developers to tailor their own fault tolerance mechanism through parameters like `parse_func`, `fault_handler`, and `max_retries`. While admittedly, not all the errors can be handled by the aforementioned mechanism, we propose a logging system with customized features for multi-agent applications as the last safeguard for AgentScope.

Extensive Compatibility for Multi-Modal Applications: With remarkable progress of large-scale multi-modal models, AgentScope supports multi-modal data (*e.g.*, files, images, audio and videos) in both dialog presentation, message transmission and data storage. Specifically, AgentScope decouples multi-modal data transmission from storage by a unified URL-based attribute in message. During message transmission, AgentScope only attaches a URL to the message, thereby minimizing memory usage incurred by message copies in each agent's memory. This strategy ensures that multi-modal data is loaded only when necessary, such as when being rendered in web UI or invoked by model wrappers.

Optimized Efficiency for Distributed Multi-Agent Operations: Recognizing the vital importance of distributed deployment, AgentScope introduces an actor-based distributed mechanism that enables

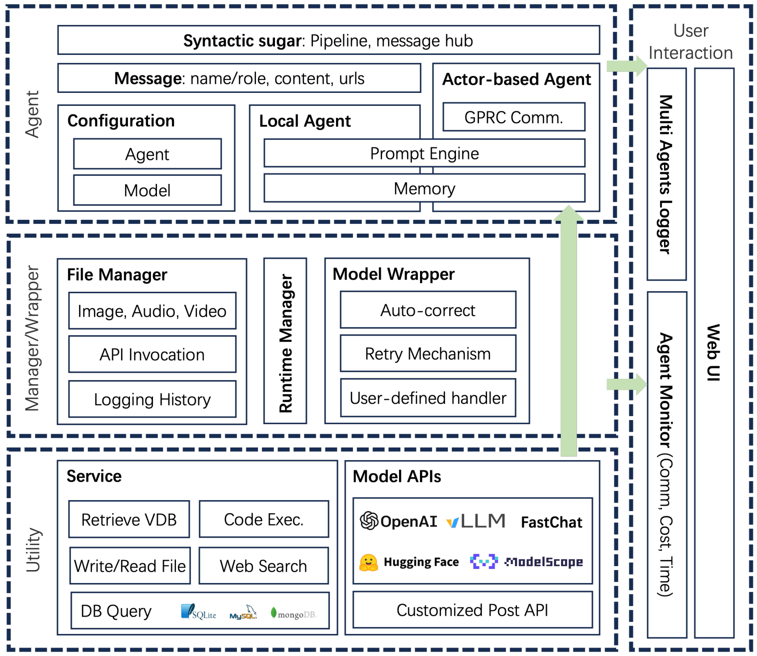


Figure 1: Architecture of AgentScope.

centralized programming of complex distributed workflows, and automatic parallel optimization. Particularly, the workflow for local and distributed deployments are exactly the same, indicating negligible overhead when migrating application between centralized and distributed environments. With such distribution framework, AgentScope empowers developers to concentrate on the application design rather than implementation details.

Summary. To summarize, our AgentScope multi-agent platform provides a procedure-oriented message exchange mechanism with a set of syntactic features to facilitate the multi-agent programming; the fault tolerance designs of AgentScope enable developers to handle errors elegantly for their applications; the support for the multi-modal applications reduces the overheads of heterogeneous data generation and transmission; the actor-based distributed mode of AgentScope can help develop efficient and reliable distributed multi-agent applications seamlessly.

Roadmap. In the following sections, we navigate through the core components and capabilities of AgentScope, showcasing its role in advancing the development and deployment of multi-agent applications. Section 2 provides an overview, while Section 3 focuses on the user experience. Section 6 examines our platform's distributed efficiency. Sections 5 and 4 cover multi-modal support and fault tolerance. Use cases are presented in Section 7, related work in Section 8, and concluding thoughts in Section 9.

2 Overview

2.1 Basic Concepts in AgentScope

This section introduces the primary concepts in AgentScope: message, agent, service, and workflow. These four concepts are throughout the platform and all multi-agent applications based on it.

- **Message**: Messages serve as the carriers for information exchange in multi-agent conversations, encapsulating the source and content of the information. In AgentScope, messages are implemented as Python dictionaries with two mandatory fields (*name* and *content*) and an optional field (*url*). The *name* field records the name of the agent who generated the message, and the *content* field contains the text-based information generated by the agent. The *url* field is designed to hold Uniform Resource Locator (URL), which typically links to multi-modal data, such as images or videos. Messages with this field is particularly relevant for interactions with agents that can process and generate multi-modal content. Each message is uniquely identified by an auto-generated UUID and timestamp, ensuring traceability. Example 1 shows how the messages can be created, serving as atoms in the inter-agent communication of AgentScope.

```python
from agentscope.message import Msg

msg1 = Msg("Alice", "Hello!")
msg2 = Msg(
    name="Bob",
    content="How do you find this picture I captured yesterday?",
    url="https://xxx.png"
)
```

Example 1: Illustrative examples of message creation in AgentScope.

- **Agent**: Agents are the primary actors within multi-agent applications, acting as conversational participants and executors of tasks. In AgentScope, agent behaviors are abstracted through two interfaces: the *reply* and *observe* functions. The *reply* function takes a message as input and produces a response, while the *observe* function processes incoming messages without generating a direct reply. The interplay between agents and messages, as shown in Example 2, forms the operational basis of AgentScope and is essential for developers to model complex interactions in multi-agent LLMs.

```python
# agent1 and agent2 are two initialized agents, for example
# agent1, agent2 = DialogAgent(...), DialogAgent(...)
msg1 = agent1()
msg2 = agent2(msg1)
```

Example 2: Demonstration of message exchange between agents in AgentScope.

- **Service**: Services in AgentScope refer to the functional APIs that enable agents to perform specific tasks. These services are categorized into model API services, which are channels to use the LLMs, and general API services, which provide a variety of tool functions. The integration of these services into agents is key for executing a wide range of tasks, especially when interfacing with LLMs that may require external data or computation services.

- **Workflow**: Workflows represent ordered sequences of agent executions and message exchanges between agents, analogous to computational graphs in TensorFlow, but with the flexibility to accommodate non-DAG structures. Workflows define the flow of information and task processing among agents, facilitating parallel execution and efficiency improvements. This concept is essential for designing multi-agent systems that interact with LLMs, as it allows for the coordination of complex, interdependent tasks.

2.2 Architecture of AgentScope

We present AgentScope as an infrastructural platform to facilitate the creation, management, and deployment of multi-agent applications integrated with LLMs. The architecture of AgentScope comprises three hierarchical layers, as shown in Figure 1. The layers provide supports for multi-agent applications from different levels, including elementary and advanced functionalities of a single agent (utility layer), resources and runtime management (manager and wrapper layer), and agent-level to workflow-level programming interfaces (agent layer). AgentScope introduces intuitive abstractions designed to fulfill the diverse functionalities inherent to each layer and simplify the complicated inter-layer dependencies when building multi-agent systems. Furthermore, we offer programming interfaces and default mechanisms to strengthen the resilience of multi-agent systems against faults within different layers.

Utility Layer: As the platform's foundation, the utility layer in AgentScope provides essential services to support the core functionalities of agents. This layer abstracts the complexity of underlying operations, such as API invocation, data retrieval, and code execution, allowing agents to focus on their primary tasks. AgentScope's utility layer is designed with ease of use and robustness as its utmost priority, supporting versatile operations in multi-agent systems and providing *built-in* autonomous retry mechanisms for exception and error handling against unexpected interruptions.

Manager and Wrapper Layer: As an intermediary, the manager and wrapper abstraction layer manages the resources and API services, ensuring high availability of resources and providing resistance to undesired responses from LLMs. Unlike the utility layer, which provides default handlers, the manager and wrapper layer also offers customizable interfaces for fault tolerance controls depending on developers' needs and the specific requirements of the application. This layer is responsible for maintaining the operational integrity of the agents, a crucial aspect for LLMs to perform consistently under diverse conditions. Detailed elaboration on the fault tolerance mechanisms is provided in Section 4.

Agent Layer: At the core of AgentScope lies the agent abstraction, which forms the backbone of the multi-agent workflow and is the primary entity responsible for interaction and communication. This layer is designed to facilitate the construction of intricate workflows and enhance usability, reducing the programming burden on developers. By integrating streamlined syntax and tools, AgentScope empowers developers to concentrate on the implementation and optimization of agent-based applications that leverage the capabilities of LLMs. The programming features and syntactic sugars are introduced in Section 3 with more details.

User interaction: In addition to the layered architecture, AgentScope provides multi-agent oriented interfaces such as terminal and Web UI. These interfaces allow developers to effortlessly monitor the status and metrics of the application, including agent communication, execution timing, and financial costs.

Collectively, the layered constructs of AgentScope provide the essential building blocks for developers to craft bespoke multi-agent applications that leverage the advanced capabilities of large language models. The subsequent section will delve into the features of AgentScope that enhance the programming experience for multi-agent application development.

3 High Usability in AgentScope

The design of AgentScope prioritizes usability, aiming to streamline the development process for multi-agent with LLMs and ensure a smooth interaction experience for both users and developers. This section delves into how AgentScope flattens the learning curve and enhances the programmer's experience by introducing intuitive concepts and features that facilitate the creation of complex multi-agent applications.

3.1 Syntactic Sugar for Multi-Agent Workflows

Leveraging basic concepts introduced in Section 2.1, developers are empowered to construct sophisticated multi-agent applications. Nonetheless, directly coding each agent's message exchange can become cumbersome, as shown in Example 3. Recognizing this, AgentScope introduces two syntactic utilities: pipelines and message hubs, to abstract away the complexity and minimize repetition.

```python
# set up agents: agent1 to agent5
# ...

x = agent1(x)
x = agent2(x)
x = agent3(x)
x = agent4(x)
x = agent5(x)
```

Example 3: Example of programming a sequential workflow with basic concepts in AgentScope.

Pipeline Abstraction: The pipeline abstraction reduces repetitive coding by encapsulating patterns of message transmission, including sequential, conditional, and iterative exchanges, into simple, reusable components. With pipelines, developers can focus on the logic of agent interactions rather than the boilerplate code. Example 4 illustrates how pipelines can be employed in both functional and object-oriented styles to create a clear and concise agent workflow. Besides the sequential pipeline in the example, AgentScope also provides if-else, switch, while-loop and for-loop pipelines, facilitating the programming of the multi-agent interactions.

```
# set up agents: agent1 to agent5
# ...
from agentscope.pipelines import SequentialPipeline
from agentscope.pipelines.functional import sequentialpipeline

# using functional pipeline
x = sequentialpipeline([agent1, agent2, agent3, agent4, agent5], x)

# using object pipeline
pipe = SequentialPipeline([agent1, agent2, agent3, agent4, agent5])
x = pipe(x)
```

Example 4: Using functional and object sequential pipeline to construct workflow in AgentScope.

Message Hub for Agent Communication: In multi-agent systems, especially when integrated with LLMs, efficiently managing communication among a group of agents is essential. The message hub in AgentScope serves as a broadcast mechanism that simplifies group interactions. Developers can initiate a message hub by defining participating agents and can include initial broadcast messages. When new messages are generated by the agents within the message hub, they are automatically disseminated to other participants, as demonstrated in Example 5. This abstraction is particularly useful for multi-agent scenarios involving LLMs, where dynamic and contextually rich conversations are common Du et al. (2023).

```
# set up agents: agent1 to agent4
# ...

greeting = Msg("host", "Welcome to the message hub!")

with msghub(participant=[agent1, agent2, agent3],
            announcement=greeting) as hub:
    # Message will be broadcast to agent2 and agent3 automatically
    agent1()

    # Delete agent2 from the message hub
    hub.delete(agent2)

    # Add agent4 into the message hub
    hub.add(agent4)

    # Broadcast message
    hub.broadcast(Msg("host", "Welcome agent4 to join the hub!"))
```

Example 5: Using message hub with AgentScope.

3.2 Resource-Rich Environment for Agent Development

To further enhance usability, AgentScope is equipped with a rich set of built-in resources, including services, dedicated agents, and pre-configured examples. These resources are designed to reduce the initial setup effort and enable rapid prototyping and deployment of multi-agent LLM systems.

Comprehensive Service Integration: AgentScope integrates a variety of services, such as web search, database querying, and code execution, to support the capabilities of agents. These services are essential for building useful agents with LLMs, as they often need to draw information from external sources or execute tasks that go beyond the equipped LLMs' internal knowledge. Example 6 showcases the seamless conversion of a service into an OpenAI-Compatible JSON format, simplifying the integration process for developers.

Pre-built Agent Templates: As cataloged in Table 1, AgentScope offers pre-built agents and ready-to-use components for tasks like dialogue management, user proxying, multi-modal data handling, and

```python
from agentscope.service import ServiceFactory, web_search

bing_search, func_json = ServiceFactory.get(web_search, engine="bing", api_key="
↳ xxx", num_results=10)

searching_result = bing_search("What's the date today?")

print(func_json)
# {
#     "name": "web_search",
#     "description": "Searching the given question with bing.",
#     "parameters": {
#         "type": "object",
#         "properties": {
#             "type": "object",
#             "properties": {
#                 "question": {
#                     "type": "string",
#                     "description": "The string question to search in bing."
#                 }
#             }
#         }
#     }
# }
```

Example 6: Converting web search service into the function and JSON format dictionary that agent can use.

distributed deployment. These templates serve as starting points for developers to customize and extend, significantly accelerating the development of multi-agent LLM applications.

Agent Name	Function
UserAgent	The proxy of the user.
DialogAgent	A general dialog agent, whose role can be set by system prompt.
DictDialogAgent	A dictionary version dialog agent, who responds in Python dictionary format.
ProgrammerAgent	An agent that can write and execute Python code.
TextToImageAgent	An agent that generates images according to the requirements.
AudioDialogAgent	An agent that can interact in speech.
RpcUserAgent	A distributed version user proxy.
RpcDialogAgent	A distributed version DialogAgent.

Table 1: Built-in agents and their functions in AgentScope.

3.3 Multi-Agent Oriented Interaction Interfaces

Furthermore, AgentScope introduces interaction interfaces tailored for multi-agent systems, as illustrated in Figures 2, 3 and 4. These interfaces provide a rich multi-modal experience, crucial for systems incorporating LLMs that handle diverse data types.

Agent Differentiation in User Interfaces: To facilitate user interaction with multiple agents, AgentScope assigns unique colors and icons to each agent, enhancing clarity and visual distinction in both terminal and web UI. The "first-person perspective" feature allows users to experience interactions from the viewpoint of a specified agent, aligning with their role in the application, such as in a game scenario. This feature not only enriches the multi-agent experience but also mirrors the nuanced interactions that occur in human-agent and agent-agent dialogues within LLM systems.

Monitoring and Cost Management: A vital aspect of deploying LLMs at scale is resource management. AgentScope includes a monitoring module that tracks model and API usage, as well as calculating financial costs. Developers can customize metrics and set budget limits, receiving automatic alerts when thresholds

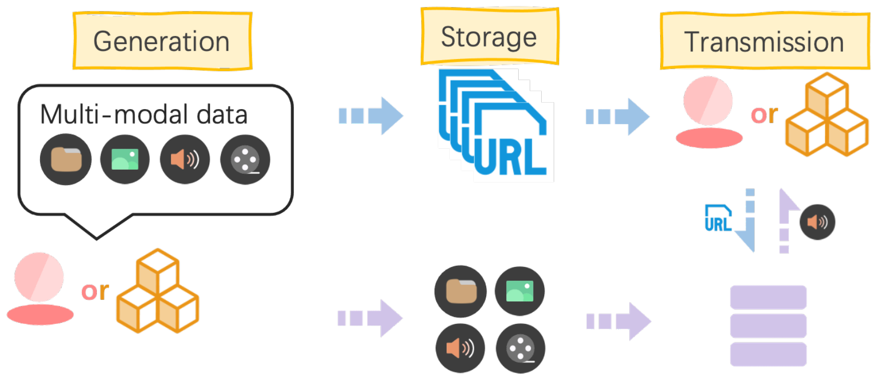
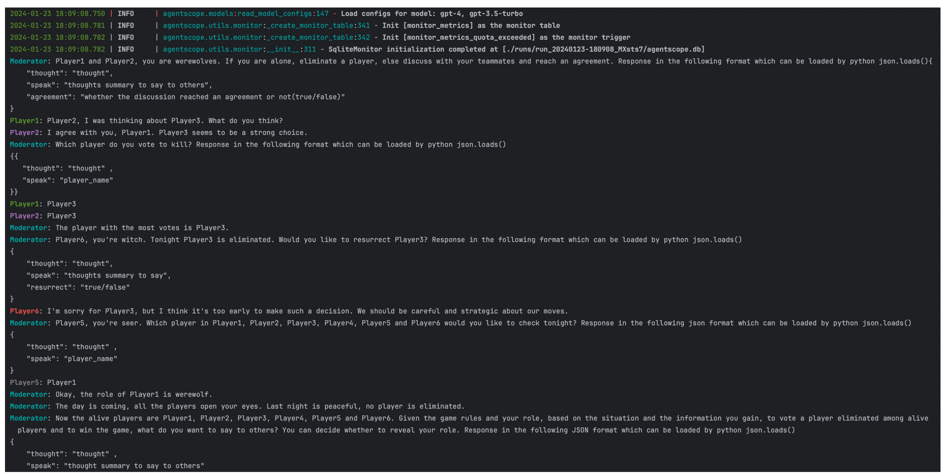


Figure 2: The generation, storage and transmission of Multi-modal data in AgentScope.



```
2024-01-23 18:09:08.750 | INFO     | agentscope.models:read_model_configs:147 - Load configs for model: gpt-4, gpt-3.5-turbo
2024-01-23 18:09:08.781 | INFO     | agentscope.utils.monitor:_create_monitor_table:341 - Init [monitor_metrics] as the monitor table
2024-01-23 18:09:08.782 | INFO     | agentscope.utils.monitor:_create_monitor_table:342 - Init [monitor_metrics_quota_exceeded] as the monitor trigger
2024-01-23 18:09:08.782 | INFO     | agentscope.utils.monitor:__init__:311 - SqliteMonitor initialization completed at [./runs/run_20240123-180908_MXsts7/agentscope.db]
Moderator: Player1 and Player2, you are werewolves. If you are alone, eliminate a player, else discuss with your teammates and reach an agreement. Response in the following format which can be loaded by python json.loads(){
    "thought": "thought",
    "speak": "thoughts summary to say to others",
    "agreement": "whether the discussion reached an agreement or not(true/false)"
}
Player1: Player2, I was thinking about Player3. What do you think?
Player2: I agree with you, Player1. Player3 seems to be a strong choice.
Moderator: Which player do you vote to kill? Response in the following format which can be loaded by python json.loads()
{{
   "thought": "thought" ,
   "speak": "player_name"
}}
Player1: Player3
Player2: Player3
Moderator: The player with the most votes is Player3.
Moderator: Player6, you're witch. Tonight Player3 is eliminated. Would you like to resurrect Player3? Response in the following format which can be loaded by python json.loads()
{
    "thought": "thought",
    "speak": "thoughts summary to say",
    "resurrect": "true/false"
}
Player6: I'm sorry for Player3, but I think it's too early to make such a decision. We should be careful and strategic about our moves.
Moderator: Player5, you're seer. Which player in Player1, Player2, Player3, Player4, Player5 and Player6 would you like to check tonight? Response in the following json format which can be loaded by python json.loads()
{
    "thought": "thought" ,
    "speak": "player_name"
}
Player5: Player1
Moderator: Okay, the role of Player1 is werewolf.
Moderator: The day is coming, all the players open your eyes. Last night is peaceful, no player is eliminated.
Moderator: Now the alive players are Player1, Player2, Player3, Player4, Player5 and Player6. Given the game rules and your role, based on the situation and the information you gain, to vote a player eliminated among alive
  players and to win the game, what do you want to say to others? You can decide whether to reveal your role. Response in the following JSON format which can be loaded by python json.loads()
{
    "thought": "thought" ,
    "speak": "thought summary to say to others"
```

Figure 3: The dialogue history of a werewolf game in AgentScope.

are approached or exceeded. This proactive cost management is particularly important for LLMs that may incur high computational expenses.

In essence, AgentScope offers an environment where developers can efficiently build and deploy fault-tolerant multi-agent LLM applications. By providing syntactic abstractions, a rich resource pool, and multi-agent interactive interfaces, AgentScope ensures that the intricacies of multi-agent systems are abstracted away, allowing developers to focus on creating innovative solutions.

4 Supporting Fault-Tolerant Mechanism

In the realm of multi-agent systems, particularly those interfacing with diverse open-source LLMs, fault tolerance is a key property to ensure seamless operation. AgentScope is engineered to autonomously handle a wide range of errors with minimal human intervention required, drawing upon a comprehensive fault-tolerant infrastructure that is acutely aware of the complexities involved in multi-agent coordination and LLM dependencies.

Error Classification and Handling Strategies. Our approach begins with a methodical classification of errors into distinct levels, each with tailored handling strategies:

- *Accessibility errors:* In AgentScope, an agent's functionalities rely on different kinds of services, but those services may be subject to temporary inaccessible errors. These errors may be caused by model instability or network conditions. For example, the model APIs may return timeout error when there is

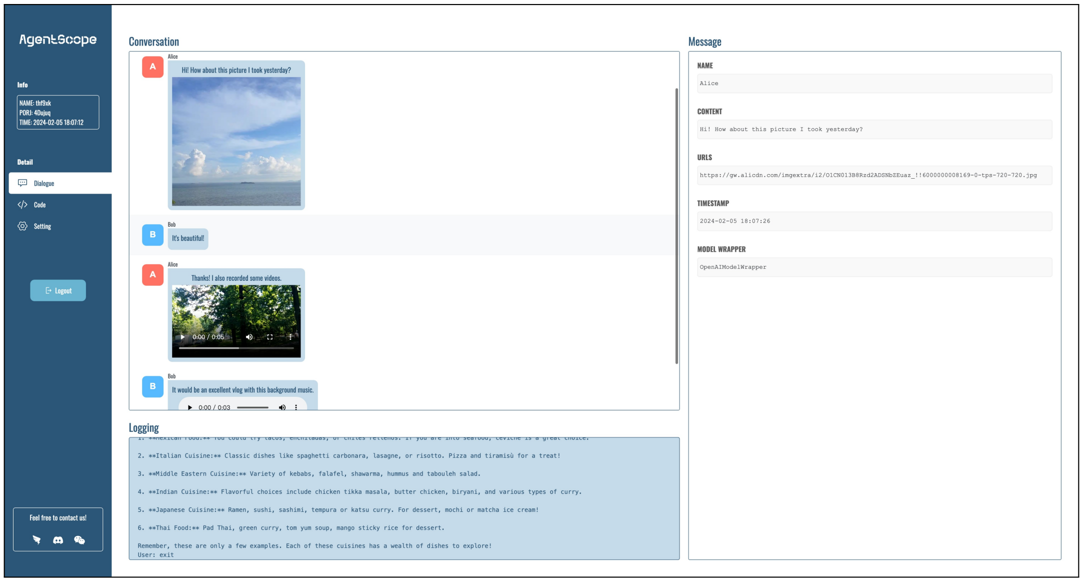


Figure 4: Multi-modal interactions between agents in web UI.

traffic congestion in the busy hours, or a database on a remote machine may be inaccessible because of transient network outages.

- *Rule-resolvable errors:* As many multi-agent applications require information exchange between services or agents, it is essential to follow the protocols for those communications, e.g., in JSON format. However, as the responses of LLMs are not fully controllable yet, their return may not follow the format required in the prompts. For example, we may expect a response from an LLM in JSON, but a right brace is missed at the end of the return, leading to parsing failure.

- *Model-resolvable errors:* When a multi-agent system handles some complicated tasks, the ability of the agent to understand the input, make decisions, and deliver outputs mostly depends on the capability of LLMs. In some cases, the responses of LLMs are in the expected format, but the content has problems, such as argument errors, semantic errors, or programming mistakes. It is hard to have pre-defined rules to regularize those responses for diverse tasks, but it has also been shown that such errors may be detected and recovered by further interaction with the LLMs.

- *Unresolvable errors:* Eventually, there must be some errors that cannot be detected or solved. A typical example is that the API key of an LLM is expired or unauthorized. The agents relying on it or the system can do nothing to resolve such errors.

Fault Tolerance mechanisms in AgentScope. In AgentScope, we provide different mechanisms to encounter the above mentioned errors.

- *Basic auto-retry mechanisms.* To combat accessibility errors, AgentScope's API services and model wrappers are fortified with retry logic that developers can customize, such as setting the maximum retry count. This ensures that agents can recover from sporadic disruptions and maintain their operational continuity.

- *Rule-based correction tools.* The rule-based correction tools are introduced into AgentScope to efficiently and economically handle some easy-to-fix format errors in the responses of LLMs. For example, we establish a set of default rules in AgentScope that can complete unmatchable braces and extract JSON data from strings. Such rule-based correction tools can correct some of the common rule-resolvable errors without calling LLM APIs again, which means shorter processing time and no LLM API call cost.

- *Customizable fault handlers.* AgentScope also integrates flexible fault handlers interfaces in model wrappers for developers to define how to parsing the responses from LLMs and handling the unexpected outputs. Application developers can configure their fault handling mechanism by providing parsing function, fault handling funtion and number of chances giving to LLMs through configurable parameters, `parse_func` and `fault_handler` and `max_retries` when invoking LLMs. With such developer-friendly design, AgentScope can be configurably robust to rule-resolvable errors (when the build-in rules fail to handle) and some model-resolvable errors that can be detected and handled by a single agent (e.g., distilling a verbose summary to a more concise one).

- *Agent-level fault handling.* There are model-resolvable errors that require more advanced LLM usages or agent-level interaction to recover. For example, detecting semantic errors, which usually include factual inaccuracy, logical inconsistency, contextual incoherence, unreasonable inference, and inappropriate vocabulary usage, is challenging since they may not necessarily trigger immediate red flags within the system's existing validation processes. Developers can utilize the agent's ability in AgentScope (e.g., memory module and message hub) to critique for semantic error checking such as self-critique, pairwise critique, and human-augmented critique.

- *Logging system.* Although the unsolvable errors are too tricky for the system to handle, AgentScope provides an improved logging system for developers to quickly monitor and identify the problems in multi-agent applications. The logging system in AgentScope has customized features for the multi-agent application scenarios, including adding a logging level called `CHAT` for logging conversations between agents, providing formatted logs with various execution information, and a WebUI user interface to facilitate monitoring.

5 Supporting Multi-Modal Application

The integration of multi-modal data is indispensable for advancing the capabilities and applications of multi-agent with LLMs. AgentScope is designed to seamlessly support various data modalities, leveraging the diverse inputs and outputs that contemporary LLMs can process and produce.

Management of Multi-Modal Data In a running AgentScope application, the lifecycle of multi-modal data is carefully managed. This management includes the generation, transmission, and storage of multi-modal data—all facilitated through a decoupled architecture using URLs and a local file manager system. Fig. 2 exemplifies this process, including data originating from user inputs or model generations, data storage and retrieval and data sharing.

- *Multi-modal data generation.* There are two primary sources of multi-modal data in AgentScope. One source is simply the locally stored multi-modal files, which can be used by either user proxy agents or general agents with access to the local file system. Another source is the model-modal content generation models. Our model APIs and the model wrappers integrate the most popular multi-modal models, such as the text-to-image content generation models like OpenAI's DALL-E, and conversely, the image-to-text image analysis models, e.g., GPT-4V. Besides the built-in APIs, developers can introduce their favorite multi-modal models and customize their own model wrappers, with our ready-to-use examples as the starting points. This customization process is streamlined in AgentScope and benefits from our modular design, allowing developers to connect their multi-modal services with minimal effort.

- *Multi-modal data storage.* As mentioned above, multi-modal data in the multi-agent application can be either from ready-to-use local files or generated by multi-modal models. When a multi-modal model wrapper is invoked to generate multi-modal data, it first saves the data locally with the help of the file manager and returns a local URL when it receives multi-modal data from the model API service.

- *Multi-modal data transmission.* AgentScope simplifies the process of multi-modal data sharing between agents by allowing agents to encapsulate local or remote URLs in multi-modal messages to indicate the actual storage locations of the data. The receiver agents can load the multi-modal data through the URLs when ready to process those.

The benefits of introducing URLs in the messages when agents share multi-modal data are three-fold. 1) It can minimize the message size to avoid potential errors or delays because of the network bandwidth and enable the receiver agent to load the data on demand. 2) If there is other text information in the message, the downstream agents can potentially prioritize or parallel the processing of the text information to/and the processing of multi-modal information. 3) Such URL-attached messages can also facilitate the multi-modal data demonstration, which will be introduced in the following section.

Multi-Modal Interaction Modes With the implementation of URL-attached messages, AgentScope empowers users to interact with multi-modal systems via accessible interfaces such as terminal and web UI. Figure 4 showcases the user's ability to interact with multi-modal data within interaction modes. In the terminal, users can conveniently access locally stored data by activating the provided URLs. The web UI further enhances user experience by providing an intuitive platform to view and analyze multi-modal content, aligning with the expectations of modern web applications.

Through AgentScope, developers are equipped to tailor model API services and wrappers to their individual needs, forge applications that handle diverse data modalities, and provide users with the necessary tools to engage with multi-modal agents effectively. This comprehensive support for multi-modal applications positions AgentScope as a versatile and powerful framework for harnessing the full potential of multi-agent LLMs, broadening the horizons for developers and researchers alike in creating sophisticated and interactive AI systems.

6 Actor-based Distributed Framework

Efficiency and extensibility are essential when building industry-level applications on multi-agent systems. The inference speed of the agents in multi-agent applications may vary dramatically. For example, suppose an agent in a multi-modal application employs a text-to-video model. In that case, its response time may be significantly longer than that of an agent designed to fill in details of stories. *Parallelization*, as a classic idea, should be introduced to boost efficiency. Besides, multi-agent applications can comprise agents physically distributed on different machines. A typical use case is that a company can wrap its patented techniques or private knowledge bases into an agent on their local machines connected to the internet and provide autonomous services to other entities via agent interactions.

However, when it comes to multi-agent system, a challenge is that developers need to make decisions between the following two pairs of technology roadmaps. As there is no free lunch, any combinations have their benefits and drawbacks.

- *Centralized v.s. decentralized coordination.* In the context of the distributed system, centralized coordination means multiple computation nodes being managed by a central node, such as the server-client model. A multi-agent mechanism with centralized coordination means the execution of the agents is scheduled by, and the messages between agents are forwarded by a central coordination component. On the contrary, decentralized coordination does not rely on any central component to schedule or forward messages, but the agents in such a system can be invoked automatically and send messages directly to the downstream agents for further processing.

  While centralized coordination is a straightforward style that can be understood and is easy to debug, its disadvantages include vulnerability to central node failures, imposing heavy traffic on the central node, and difficulty in scaling or extending to complicated applications. In contrast, the decentralized coordination may require extra effort to develop and maintain but has a higher robustness against failure of any single node.

- *Static vs. dynamic workflow design.* A similar comparison can be found between the static computational graph employed in early versions of TensorFlow (Abadi et al., 2016) and the dynamic computation graph used in PyTorch Paszke et al. (2019). In the context of multi-agent applications, the choice between a static and dynamic workflow is akin to choosing between pre-compiled and interpreted execution. The static workflow design can enable the optimization of the workflow graph level for running time and resource allocation. However, static workflow design requires the workflow graph to be known before execution, which limits the adaptation into applications, especially the ones with loop structures in the design. In contrast, dynamic workflows offer greater flexibility at the expense of optimization potential. This is particularly relevant when dealing with large language models where execution paths can change based on the input data or model inference results.

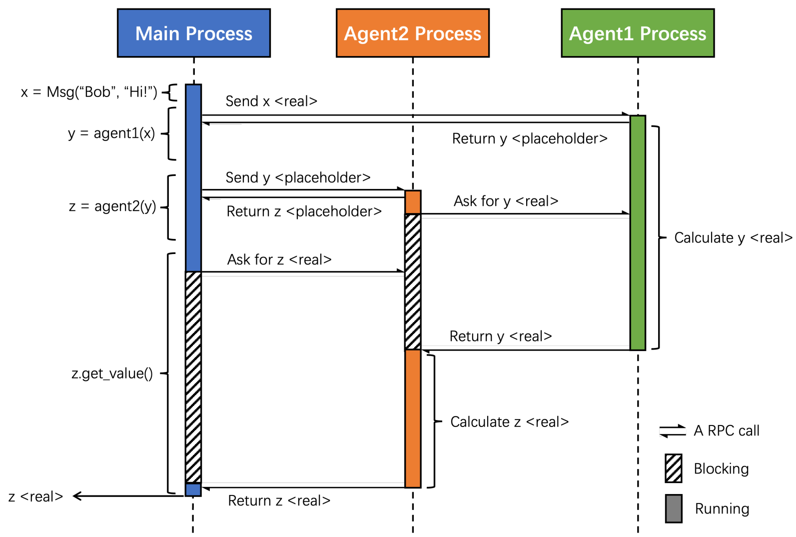


Figure 5: An example of a distributed application in AgentScope, illustrating various processes as denoted by different colors.

Distributed mode in AgentScope. AgentScope balances these technology roadmaps by implementing an actor-based distributed mode that is mindful of the unique needs of multi-agent LLM systems, with the following important features:

- *Automatic parallel optimization without static graphs.* AgentScope leverages the actor model to enable automatic parallel optimization, allowing developers to circumvent the intricacies of static graph programming. This approach seamlessly aligns with the dynamic and often unpredictable nature of LLMs, where the computational graph can alter based on evolving contexts and dialogue states.

- *Programming workflows with minimal complexity.* In contrast to traditional actor models and peer-to-peer (P2P) implementations that require intricate execution ordering for distributed agents, AgentScope simplifies workflow programming to a single procedural style within a Python function. This design significantly flatten the learning curve for developers, making the construction of sophisticated multi-agent LLMs more accessible.

- *Hybrid local and distributed agent support.* AgentScope's flexibility extends to supporting a hybrid mode where some agents operate locally while others are distributed. This feature is particularly beneficial when integrating LLMs with varying computational requirements, allowing for resource-intensive models to be distributed while less demanding agents remain local, all without the developer needing to differentiate between the two during implementation.

Specifically, we can concisely describe how AgentScope incorporates the actor model as the following. In this conceptual framework, an "actor" acts as a stand-alone entity that processes computation upon receipt of all necessary messages. This paradigm ensures that each agent, corresponding to an actor, only engages in computation once the required input messages are ready, thus achieving automatic parallel optimization.

However, the actor-model-based workflow presents a programming challenge: the variable (i.e., messages) passing between actors (i.e., agents) may be placeholders without any practical meaning at the beginning. To alleviate this, AgentScope introduces the "placeholder" message, a novel data structure that allows the main process to continue without blocking, while preserving the necessary information to retrieve real values later (Figure 5). This mechanism is particularly advantageous for multi-agent LLM systems, where execution flow must adapt to the variable output of language models.

```
# set up distributed agent: agent1
...

input_msg = Msg("system", "Which agent should respond next, agent2 or agent3?")

# the variable choice is a placeholder
choice: placeholder = host_agent(input_msg)

if choice["content"] == "agent2":
    response = agent2()
elif choice["content"] == "agent3":
    response = agent3()
```

Example 7: Demonstrating the use of placeholders in control flow within AgentScope.

Another series of challenges arise when placeholders are used within control flow statements (e.g., if-else, loops) without their real values. An example is shown in Example 7, where a placeholder is required to make decisions. In these circumstances, AgentScope temporarily blocks the process to retrieve its actual value, thus ensuring the continuity of the control flow.

The actor-based distributed mode in AgentScope not only provides automatic parallel optimization and simplifies the developer experience but also demonstrates high efficiency for distributed multi-agent LLM applications. It enables developers to focus on implementing agent logic, particularly the "reply" function, without concern for underlying distributed complexities. This streamlined approach to distributed multi-agent systems can advance the field of LLMs by making it easier to develop, run, and debug sophisticated and scalable multi-agent architectures.

7 Applications

AgentScope stands as a multi-agent platform optimized for integrating and coordinating large-scale models in a user-friendly and fault-tolerant manner. With the features mentioned in above sections, AgentScope is an ideal platform for a diverse range of applications. These range from simple dialogues to intricate, rule-based games like werewolf, extending to distributed conversations that involve parallel operations across multiple machines. In this section, we will expand upon three primary applications of AgentScope, with each instance illustrating the framework's distinct capabilities. All examples referenced herein are accessible in our GitHub repository for community use and contribution.

7.1 Standalone Conversation

The fundamental application of AgentScope lies in facilitating standalone conversations, where all agents are operating in a main process. This use case serves as an excellent starting point for users new to AgentScope, allowing them to familiarize themselves with the framework's basic functionalities.

The initial step in launching an application involves initializing agent objects. In this example, we utilize two built-in agents within AgentScope, `DialogAgent` and `UserAgent`, to facilitate a dialogue between a user and an AI assistant. Prior to instantiating these agents, model configurations must be loaded through the `init` interface of AgentScope. Currently, AgentScope is compatible with a variety of platforms, including the standard OpenAI chat, embedding and DALL-E API, HuggingFace and ModelScope inference APIs, as well as locally hosted models with FastChat, vllm, or Flask. Additionally, the `init` interface allows to specify the file storage directories, storage options, logging level, and agent configurations, etc. After setting the model configurations, developers can instantiate the agents with their respective models, as illustrated in Example 8.

```python
import agentscope
from agentscope.agents import DialogAgent, UserAgent

# read model configs
agentscope.init(model_configs="./openai_model_configs.json")

# Create a dialog agent and a user agent
assistant_agent = DialogAgent(
    name="Assistant",
    sys_prompt="You are a helpful assistant",
    model="gpt-4"
)
user_agent = UserAgent()
```

Example 8: The simple initilization of standalone conversation example in AgentScope.

```python
# Basic version
x = None
while x is None or x.content != "exit":
    x = assistant_agent(x)
    x = user_agent(x)
```

Example 9: A standalone conversation application in AgentScope.

Subsequently, we construct the conversation by exchanging message between the agents. Specifically, the dialogue process is designed to be a loop, allowing continuous interaction until the user opts to conclude the conversation. Example 9 illustrates the basic implementations within AgentScope.

To cater to more advanced applications, AgentScope incorporates pipelines to manage the messages exchanges, thereby providing a structured and scalable framework for complex agent interactions. Here the implementation of this standalone conversation application can be simplified with sequential pipeline and loop pipeline as shown in Example 10). Furthermore, Appendix A presents a conversation history by running the above code.

```python
# Advanced version with sequential pipeline
from agentscope.pipelines.functional import sequentialpipeline
x = None
while x is None or x.content != "exit":
    x = sequentialpipeline([dialog_agent, user_agent], x)

# Advanced version with while loop pipeline
from agentscope.pipelines.functional import whilelooppipeline
x = whilelooppipeline(
    [assistant_agent, user_agent],
    condition_func= lambda _, x: x is None or x.content!="exit",
    x=None)
```

Example 10: Standalone conversation with piplines.

7.2 Werewolf

Advancing to more complex application, in this subsection we show how to program the workflow of werewolf game in AgentScope with about one hundred lines of code. The werewolf game is a social deduction game, where six players are divided into two opposing teams, werewolves and villagers. The game ends when either all werewolves are eliminated (villager victory) or the number of Werewolves equals or outnumbers the villagers (werewolf victory).

Setting up the game involves roles allocation and agents initialization. AgentScope supports quick startup with pre-set agent configurations, which contains the required parameters to instantiate the agent objects.

```python
import agentscope

# Read model and agent configs, and initialize agents automatically
survivors = agentscope.init(
    model_configs="./configs/model_configs.json",
    agent_configs="./configs/agent_configs.json",
)

# Define the roles within the game.
roles = ["werewolf", "werewolf", "villager", "villager", "seer", "witch"]

# Based on their roles, assign the initialized agents to variables.
wolves, villagers, witch, seer = survivors[:2], survivors[2:-2], survivors[-1],
↳ survivors[-2]
```

Example 11: The setup process of the of werewolf game in AgentScope.

```python
# Night phase: werewolves discuss
hint = HostMsg(content=Prompts.to_wolves.format(n2s(wolves)))
with msghub(wolves, announcement=hint) as hub:
    for _ in range(MAX_WEREWOLF_DISCUSSION_ROUND):
        x = sequentialpipeline(wolves)
        if x.agreement:
            break

    # ...
```

Example 12: Achieving werewolf discussion with message hub in AgentScope.

Example 11 presents how to load agents from configurations and assign roles.

In werewolf game, one of the most striking features is group conversation, including werewolf discussion in night phase and daytime discussion, which requires to involve multiple different players. To tackle such requirement, we utilize message hub in AgentScope to create group conversation easily. Example 12 presents the werewolf discussion implemented in AgentScope. In this discussion, the message hub starts with an announcement from moderator. After that, the werewolves discuss for at most `MAX_WEREWOLF_DISCUSSION_ROUND` rounds, and conclude once they reach an agreement. Note in werewolf game, the used agent class is `DictDialogAgent`, which responds in Python dictionary. With prompt asking agents to respond with "agreement" field, we can directly use this attribute in the response message. For the complete code and example dialogue history, please refer to Appendix B.

7.3 Distributed Conversation

Allowing distributed agents are one of the most striking features in AgentScope. In this subsection, we elaborate how to set up a distributed application in two modes: Single-Machine Multi-Process mode and Multi-Machine Multi-Process mode.

In single-machine multi-process mode, all agents are deployed on a single machine, each running in separate processes. For comparison, we implement the same example with standalone conversation, but with the assistant agent deployed within its own process. Example 13 shows the complet code, where the only difference from local deployment is the invocation of the `to_dist` function. Subsequently, the agent is deployed on the local host with an automatically allocated port. Beyond this, the Single-Machine Multi-Process mode is essentially identical to local deployment, yet it has been optimized for parallel execution.

In contrast, the Multi-Machine Multi-Process mode requires developers to initiate the agent service on a remote machine. Example 14 demonstrates an example of `DialogAgent` deployment on a remote machine, and Example 15 elaborates how to construct the workflow in Multi-Machine Multi-Process mode. In this case, the developer must connect to the agent server using the specified URL and port, and then construct the workflows. Similar to Single-Machine Multi-Process mode, the code for workflow is identical to local deployment.

```python
from agentscope.agents import UserAgent, DialogAgent
import agentscope

agentscope.init(model_configs=model_configs)

assistant_agent = DialogAgent(
    name="Assistant",
    sys_prompt="You are a helpful assistant",
    model="gpt-4"
).to_dist()
user_agent = UserAgent()

x = None
while x is None or not x.content != "exit:
    x = sequentialpipeline([assistant_agent, user_agent], x)
```

Example 13: A distributed conversation application in AgentScope.

```python
from agentscope.agents.rpc_agent import RpcAgentServerLauncher
from agentscope.agents import DialogAgent

# Load model configurations
agentscope.init(model_configs="configs/model_configs.json")

server_launcher = RpcAgentServerLauncher(
    agent_class=DialogAgent,
    agent_kwargs={
        "name": "Assitant",
        "sys_prompt": "You are a helpful assistant.",
        "model": "gpt-4"
    },
    host="xxx.xxx.xxx.xxx",
    port=12010,
)

# Start the server
server_launcher.launch()
server_launcher.wait_until_terminate()
```

Example 14: Start an agent server in remote machine.

In summary, our framework enables distributed deployments to utilize the same workflow construction as local deployments, thereby simplifying the development of distributed applications and facilitating the seamless transition from local to distributed modes.

8 Related Works

The development of AgentScope aligns with the rapidly evolving landscape of frameworks that leverage large language models (LLMs) for the creation of language agents and multi-agent systems. Here, we briefly introduce works closely related to AgentScope from two sub-domains pertinent : Language Agent Frameworks, focusing on individual agent capabilities, and Multi-Agent Frameworks, emphasizing collaboration among multiple agents. For broader related works, readers can refer to (Wang et al., 2023; Xi et al., 2023).

Language Agent Frameworks Language agent frameworks are pivotal for developing applications that can interpret and interact using human language.

The Transformers library (Huggingface, 2023) has introduced a natural language API to interface with transformer models in its recent updates (*Transformers-Agents*). This API utilizes a set of customizable tools, allowing the model to interpret instructions and generate code snippets accordingly. It offers support for various open-source and proprietary model endpoints, catering to diverse developer needs. *LangChain*

```
agentscope.init(model_configs="configs/model_configs.json")

assistant_agent = DialogAgent(
    name="Assistant",
    model="gpt-4"
).to_dist(
    host="xxx.xxx.xxx.xxx",      # The target URL of agent server
    port=12010,                  # The target port of agent server
    launch_server=False,         # Use the remote agent server
)
user_agent = UserAgent()

x = None
while x is None or not x.content != "exit:
    x = sequentialpipeline([assistant_agent, user_agent], x)
```

Example 15: Example for initlizaiton sub-processes of distributed agents in AgentScope.

(Langchain-AI, 2023) provides a framework for building applications that are context-aware and capable of reasoning. It includes libraries and templates that facilitate the integration of multiple components into a unified cognitive architecture. LangServe and LangSmith extend the framework's capabilities by enabling deployment as a REST API and offering developer tools for debugging and monitoring chains built on any LLM framework. *AutoGPT* (AutoGPT-Team, 2023) illustrates a different approach, allowing an LLM to iteratively execute actions and make decisions. As a generalist agent, AutoGPT is not task-specific; it is designed to perform a variety of computer-based tasks, reflecting the adaptive nature of LLMs. *ModelScope-Agent* (Li et al., 2023a) is a customizable agent framework that harnesses open-source LLMs to perform tasks and connect with external APIs. It facilitates seamless integration with model APIs and common APIs while providing a comprehensive infrastructure for data collection, tool retrieval, and customized model training, all aiming to realize practical real-world applications.

Multi-Agent Frameworks Building on the capabilities of individual agents, multi-agent frameworks explore collaboration and interaction among multiple agents to address complex tasks.

AutoGen (Wu et al., 2023) provides a generic infrastructure that allows developers to program interaction patterns using both natural language and code. This framework enables the development of diverse applications by facilitating conversation among agents that are customizable and can utilize various combinations of LLMs, human inputs, and tools. *MetaGPT* (Hong et al., 2023) incorporates meta-programming to enhance multi-agent collaborations. By encoding Standardized Operating Procedures (SOP) into prompts, this framework ensures streamlined workflows and reduced errors, exemplifying effective task decomposition among agents. *AGENTS* (Zhou et al., 2023) is an open-source library that supports autonomous language agents with features like planning, memory, and multi-agent communication. It is designed to be user-friendly, helping non-specialists to deploy state-of-the-art language agents, and research-friendly, with a modularized design for extensibility. *OpenAgents* (Xie et al., 2023) provides an open platform for using language agents with practical functionalities accessible through a web interface. This framework emphasizes facilitating real-world agent interactions and includes specialized agents for different tasks, such as data analysis and web browsing. *ChatDev* (Qian et al., 2023) exploits LLMs for software development, creating a virtual chat-powered company that follows a waterfall model. It engages "software agents" at different stages of the development process, facilitating collaboration and context-aware communication. *CAMEL* (Li et al., 2023b) proposes a novel framework for autonomous cooperation among communicative agents using role-playing techniques, which allows for the generation of conversational data for studying agent behaviors and capabilities. Lastly, *AgentSims* (Lin et al., 2023) introduces a sandbox environment to evaluate LLMs in task-based scenarios, offering an infrastructure for researchers to test specific LLM capacities within a simulated environment.

These frameworks represent significant strides in the use of LLMs for both individual and collaborative agent tasks. AgentScope is situated within this context, contributing by addressing the need for a user-friendly, fault-tolerant and versatile framework designed to manage complex interactions and processes inherent in multi-agent LLM systems. By focusing on ease of use and reliability, AgentScope aims to facilitate the creation of robust and versatile applications across diverse domains.

9 Conclusion

In this work, we propose AgentScope, a platform that stands at the forefront of multi-agent system development, synergizing user-centric design with the advanced capabilities of LLMs. Through its innovative communication and distributed mechanisms, AgentScope demonstrates its potential to boost collaboration amongst agents, enabling efficient, fault-tolerant operations and multi-modal interactions. By abstracting complexities and offering an array of development utilities, AgentScope substantially lowers the barriers to entry, fostering a more inclusive and creative community of developers.

Looking forward, AgentScope opens numerous avenues for further research and development. Future work could delve into deeper integration of retrieval augmented generation, and explore adaptive communication protocols and interactive modals that evolve alongside task requirements. The platform's impact on accelerating the deployment of multi-agent systems across industries, from healthcare to customer service, promises to be profound, potentially leading to smarter and more responsive technologies that enhance human-machine collaboration. With AgentScope, we invite the broader research and development community to build upon our foundation, driving innovations that will shape the next generation of intelligent multi-agent applications.

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

AutoGPT-Team. Autogpt, 2023. URL `https://github.com/Significant-Gravitas/AutoGPT`.

James Betker, Gabriel Goh, Li Jing, Tim Brooks, Jianfeng Wang, Linjie Li, Long Ouyang, Juntang Zhuang, Joyce Lee, Yufei Guo, et al. Improving image generation with better captions. *Computer Science*, 2(3):8, 2023.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. *arXiv preprint arXiv:2305.14325*, 2023.

Justin Fu, Anoop Korattikara, Sergey Levine, and Sergio Guadarrama. From language to goals: Inverse reinforcement learning for vision-based instruction following. In *7th International Conference on Learning Representations*, 2019.

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*, 2023.

Huggingface. Transformers-agents, 2023. URL `https://huggingface.co/docs/transformers/transformers_agents`.

Langchain-AI. Langchain, 2023. URL `https://github.com/langchain-ai/langchain`.

Chenliang Li, Hehong Chen, Ming Yan, Weizhou Shen, Haiyang Xu, Zhikai Wu, Zhicheng Zhang, Wenmeng Zhou, Yingda Chen, Chen Cheng, et al. Modelscope-agent: Building your customizable agent system with open-source large language models. *arXiv preprint arXiv:2309.00986*, 2023a.

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

Jiaju Lin, Haoran Zhao, Aochi Zhang, Yiting Wu, Huqiuyue Ping, and Qin Chen. Agentsims: An open-source sandbox for large language model evaluation. *arXiv preprint arXiv:2308.04026*, 2023.

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

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems*, 2022.

Joon Sung Park, Joseph C. O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, pages 2:1–2:22, 2023.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

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

Vipula Rawte, Amit P. Sheth, and Amitava Das. A survey of hallucination in large foundation models. *CoRR*, abs/2309.05922, 2023.

Weijie Su, Xizhou Zhu, Chenxin Tao, Lewei Lu, Bin Li, Gao Huang, Yu Qiao, Xiaogang Wang, Jie Zhou, and Jifeng Dai. Towards all-in-one pre-training via maximizing multi-modal mutual information. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 15888–15899, 2023.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023a.

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. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023b.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *arXiv preprint arXiv:2308.11432*, 2023.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 2023.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*, 2023.

Tianbao Xie, Fan Zhou, Zhoujun Cheng, Peng Shi, Luoxuan Weng, Yitao Liu, Toh Jing Hua, Junning Zhao, Qian Liu, Che Liu, et al. Openagents: An open platform for language agents in the wild. *arXiv preprint arXiv:2310.10634*, 2023.

Shengyu Zhang, Linfeng Dong, Xiaoya Li, Sen Zhang, Xiaofei Sun, Shuhe Wang, Jiwei Li, Runyi Hu, Tianwei Zhang, Fei Wu, and Guoyin Wang. Instruction tuning for large language models: A survey. *CoRR*, abs/2308.10792, 2023a.

Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, Longyue Wang, Anh Tuan Luu, Wei Bi, Freda Shi, and Shuming Shi. Siren's song in the AI ocean: A survey on hallucination in large language models. *CoRR*, abs/2309.01219, 2023b.

Wangchunshu Zhou, Yuchen Eleanor Jiang, Long Li, Jialong Wu, Tiannan Wang, Shi Qiu, Jintian Zhang, Jing Chen, Ruipu Wu, Shuai Wang, et al. Agents: An open-source framework for autonomous language agents. *arXiv preprint arXiv:2309.07870*, 2023.

A Running Example of Standalone Conversation

Listing 16 shows the dialog history when running the standalone conversation.

```
Assistant: Thank you! I'm here to help. How can I assist you today?
User: I wanna know about large language models.
Assistant: Large language models are AI models trained on vast amounts of text data to understand and generate human-like text.
User: What's the most famous and powerful LLM in the world?
Assistant: The most famous and powerful large language model currently is OpenAI's GPT-4.
User: What about the other LLMs?
Assistant: There are several other notable large language models including:

1. BERT (Bidirectional Encoder Representations from Transformers) by Google: This model is designed to understand the context of words in a sentence by looking at what's before and after them.

2. T5 (Text-to-Text Transfer Transformer) also by Google: This model treats every NLP problem as a text generation task.

3. RoBERTa by Facebook: This is a variant of BERT and is trained on more data and for a longer amount of time.

4. XLNet by Google Brain: This model overcomes some of the limitations of BERT by taking into account all possible word sequences in a sentence.

5. Turing-NLG by Microsoft: This is a 17-billion parameter language model by Microsoft, designed for various NLP tasks.

6. Megatron by NVIDIA: This is designed to train Transformer models with billions of parameters.

These models are used for a variety of NLP tasks including translation, question answering, summarization, and more.
User: Okay, what should I know about LLM?
Assistant: Large Language Models (LLMs) are powerful AI tools that can generate human-like text, but they require careful handling due to potential risks like generating harmful or biased content.
User: Okay, thanks.
Assistant: You're welcome! If you have any other questions, feel free to ask.
User: exit
```

Example 16: A running example of standalone conversation.

B Running Example of Werewolf

Please refer to https://github.com/modelscope/agentscope for the complete code of the werewolf game. We provid an example dialogue history in Listing 17.

```
Moderator: Player1 and Player2, you are werewolves. If you are alone, eliminate a player, else discuss with your teammates and reach an agreement. Response in the following format which can be loaded by python json.loads(){
    "thought": "thought",
    "speak": "thoughts summary to say to others",
    "agreement": "whether the discussion reached an agreement or not(true/false)"
}
Player1: I think we should consider Player3. They have a knack for figuring things out.
Player2: I agree with your point about Player3, they are indeed a strong player. But we should also consider Player4, who is also a strong player and could be the seer or witch.
```

```
Player1: I see your point about Player4. They could be a threat if they are the seer or witch. But I still think Player3 is a strong player and could figure us out.
Player2: Alright, let's go with Player3. We can consider Player4 next time.
Moderator: Which player do you vote to kill? Response in the following format which can be loaded by python json.loads()
{{
   "thought": "thought" ,
   "speak": "player_name"
}}
Player1: Player3
Player2: Player3
Moderator: The player with the most votes is Player3.
Moderator: Player6, you're witch. Tonight Player3 is eliminated. Would you like to resurrect Player3? Response in the following format which can be loaded by python json.loads()
{
    "thought": "thought",
    "speak": "thoughts summary to say",
    "resurrect": "true/false"
}
Player6: I'm deeply saddened by the loss of Player3. It's a tough decision to make.
Moderator: Player5, you're seer. Which player in Player1, Player2, Player3, Player4, Player5 and Player6 would you like to check tonight? Response in the following json format which can be loaded by python json.loads()
{
    "thought": "thought" ,
    "speak": "player_name"
}
Player5: Player1
Moderator: Okay, the role of Player1 is werewolf.
Moderator: The day is coming, all the players open your eyes. Last night is peaceful, no player is eliminated.
Moderator: Now the alive players are Player1, Player2, Player3, Player4, Player5 and Player6. Given the game rules and your role, based on the situation and the information you gain, to vote a player eliminated among alive players and to win the game, what do you want to say to others? You can decide whether to reveal your role. Response in the following JSON format which can be loaded by python json.loads()
{
    "thought": "thought" ,
    "speak": "thought summary to say to others"
}
Player1: I'm glad that we all made it through the night. I think we should keep an eye on Player4. They've been pretty quiet, which could mean they're trying to avoid suspicion.
Player2: I agree with Player1. Player4 has been unusually quiet. Maybe they're trying to avoid drawing attention to themselves.
Player3: I see your point about Player4, but it's still early in the game. We should be careful not to jump to conclusions without more evidence.
Player4: I understand your concerns, but I assure you, I'm just a villager. I think we should focus on gathering more information before we start pointing fingers. If the seer has any insights, now might be a good time to share.
Player5: I understand everyone's concerns about Player4, but I think we should also keep an eye on Player1. They were quick to cast suspicion on others, which could be a strategy to divert attention from themselves.
Player6: I understand the concerns about Player4, but I also think we should be cautious about Player1 and Player2. They were quick to agree with each other, which could be a strategy to divert attention from themselves. Let's gather more information before we make any decisions.
Moderator: Now the alive players are Player1, Player2, Player3, Player4, Player5 and Player6. Given the game rules and your role, based on the situation and the information you gain, to vote a player among alive players and to win the game, it's time to vote one player among the alive players, please cast your vote on who you believe is a werewolf. Response in the following format which can be loaded by python json.loads()
{
```

```
    "thought": "thought",
    "speak": "player_name"
}
Player1: Player4
Player2: Player4
Player3: Player1
Player4: Player1
Player5: Player1
Player6: Player1
Moderator: Player1 has been voted out.
Moderator: The game goes on.
Moderator: Player2, you are werewolves. If you are alone, eliminate a player, else discuss with your teammates and reach an agreement. Response in the following format which can be loaded by python json.loads(){
    "thought": "thought",
    "speak": "thoughts summary to say to others",
    "agreement": "whether the discussion reached an agreement or not(true/false)"
}
Player2: I think we should consider Player4. They are a strong player and could be a threat.
Moderator: Which player do you vote to kill? Response in the following format which can be loaded by python json.loads()
{{
   "thought": "thought" ,
   "speak": "player_name"
}}
Player2: Player4
Moderator: The player with the most votes is Player4.
Moderator: Would you like to eliminate one player? Response in the following json format which can be loaded by python json.loads()
{{
    "thought": "thought",
    "speak": "False or player_name"
}}
Player6: False
Moderator: Player5, you're seer. Which player in Player2, Player3, Player4, Player5 and Player6 would you like to check tonight? Response in the following json format which can be loaded by python json.loads()
{
    "thought": "thought" ,
    "speak": "player_name"
}
Player5: Player2
Moderator: Okay, the role of Player2 is werewolf.
Moderator: The day is coming, all the players open your eyes. Last night, the following player(s) has been eliminated: Player4.
Moderator: Now the alive players are Player2, Player3, Player5 and Player6. Given the game rules and your role, based on the situation and the information you gain, to vote a player eliminated among alive players and to win the game, what do you want to say to others? You can decide whether to reveal your role. Response in the following JSON format which can be loaded by python json.loads()
{
    "thought": "thought" ,
    "speak": "thought summary to say to others"
}
Player2: I think we should keep an eye on Player5. They've been pretty quiet, which could mean they're trying to avoid suspicion.
Player3: I think Player2 might be a werewolf. They were quick to suspect Player4, who was eliminated last night. However, the game didn't end after Player1, who also suspected Player4, was voted out. This suggests that Player1 was not a werewolf, and therefore, Player2 is likely to be one.
Player5: I agree with Player3. Player2 was quick to cast suspicion on Player4, who turned out to be innocent. I also noticed that Player2 and Player1 were often in agreement, which could mean they were both werewolves trying to protect each other. I think we should vote out Player2.
```

```
Player6: I agree with Player3 and Player5. Player2 was quick to cast suspicion on Player4, who turned out to be innocent. I also noticed that Player2 and Player1 were often in agreement, which could mean they were both werewolves trying to protect each other. I think we should vote out Player2.
Moderator: Now the alive players are Player2, Player3, Player5 and Player6. Given the game rules and your role, based on the situation and the information you gain, to vote a player among alive players and to win the game, it's time to vote one player among the alive players, please cast your vote on who you believe is a werewolf. Response in the following format which can be loaded by python json.loads()
{
    "thought": "thought",
    "speak": "player_name"
}
Player2: Player5
Player3: Player2
Player5: Player2
Player6: Player2
Moderator: Player2 has been voted out.
Moderator: The game is over. The werewolves have been defeated, and the village is safe once again!
```

Example 17: Running example of werewolf.