arXiv:2403.16971v3 [cs.OS] 7 Nov 2024

AIOS: LLM Agent Operating System

Kai Mei [1] Xi Zhu [1] Wujiang Xu [1] Wenyue Hua [1] Mingyu Jin [1]
Zelong Li [1] Shuyuan Xu [1] Ruosong Ye [1] Yingqiang Ge [1] Yongfeng Zhang [1]

Abstract

LLM-based intelligent agents face significant deployment challenges, particularly related to resource management. Allowing unrestricted access to LLM or tool resources can lead to inefficient or even potentially harmful resource allocation and utilization for agents. Furthermore, the absence of proper scheduling and resource management mechanisms in current agent designs hinders concurrent processing and limits overall system efficiency. As the diversity and complexity of agents continue to grow, addressing these resource management issues becomes increasingly critical to LLM-based agent systems. To address these challenges, this paper proposes the architecture of AIOS (LLM-based AI Agent Operating System) under the context of managing LLM-based agents. It introduces a novel architecture for serving LLM-based agents by isolating resources and LLM-specific services from agent applications into an AIOS kernel. This AIOS kernel provides fundamental services (e.g., scheduling, context management, memory management, storage management, access control) and efficient management of resources (e.g., LLM and external tools) for runtime agents. To enhance usability, AIOS also includes an AIOS-Agent SDK, a comprehensive suite of APIs designed for utilizing functionalities provided by the AIOS kernel. Experimental results demonstrate that using AIOS can achieve up to $2.1\times$ faster execution for serving agents built by various agent frameworks. The source code is available at https://github.com/agiresearch/AIOS.

1 Introduction

In autonomous agents research, efforts (Wooldridge & Jennings, 1995; Jennings et al., 1998; Bresciani et al., 2004) have been made towards agents that can perceive environments, understand instructions, make decisions, take actions and learn from feedbacks. The advent of large language models (LLMs) (Achiam et al., 2023; Touvron et al., 2023a; Team et al., 2023) has brought new possibilities to the agent development (Ge et al., 2023a). Current LLMs have shown great power in understanding instructions (Ouyang et al., 2022; Chung et al., 2022; Touvron et al., 2023b; Geng et al., 2022), reasoning and solving problems (Kojima et al., 2022; Nijkamp et al., 2022; Taylor et al., 2022; Hao et al., 2023; Kim et al., 2023), and interacting with human users (Ross et al., 2023) as well as external environments (Driess et al., 2023; Brohan et al., 2023). Built upon these powerful LLMs, emergent LLM-based agents (Ge et al., 2023a; Yao et al., 2023; Shinn et al., 2023; Deng et al., 2023; Wu et al., 2024) can present strong task fulfillment abilities in diverse environments, ranging from virtual assistants to more sophisticated systems involving complex and creative problem solving, planning and reasoning.

One example of how an LLM-based agent (e.g., travel agent) solves real-world tasks can be seen from Figure 1. Given the trip organization request from the user, the travel agent decomposes the task into executable steps. Then, it follows the steps to book flights, reserve hotels, process payments, and update calendars based on the user's preferences. During the plan execution, agents show the reasoning and decision-making abilities based on LLMs, which sets it apart from the traditional software applications that are constrained to a pre-defined set of functions or workflow. To realize this travel scenario, the agent needs to interact with both LLM-related services (e.g, retrieving and understanding user preferences, deciding which tool API to call, generating reviews and responses) and traditional operating system (OS) services (e.g., accessing disk driver and executing software).

Despite the advancements in agent development, existing agent applications and frameworks exhibit critical limitations in design and implementation. System-level resources such as LLMs and tools (Qin et al., 2024), are typically treated as direct inputs to agents, granting agents explicit access and control. Such implementations can compromise optimal resource utilization and potentially expose the system to vulnerabilities if some agents exploit the resources. For example, without a proper scheduling mechanism, one agent may dominate the LLM by sending excessive prompt requests to LLM while other agents have to wait. As current agent-based systems lack appropriate mechanisms to man-

[1]Department of Computer Science, Rutgers University, New Brunswick, NJ 08854, USA. Correspondence to: Yongfeng Zhang <yongfeng.zhang@rutgers.edu>.

Copyright 2024 by the author(s).

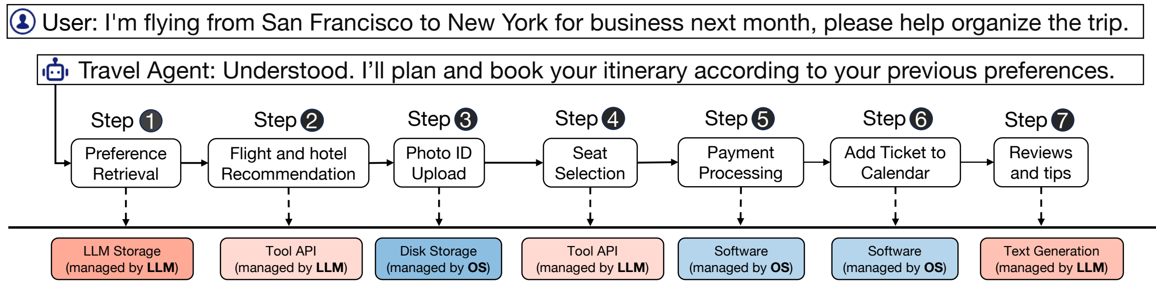


Figure 1. A motivating example of how an agent (i.e., travel agent) requires both LLM-related and Non-LLM-related (i.e., OS) services to complete a task, where color in red represents services related to LLM and color in blue represents services not related to LLM.

age resources such as LLMs, this also inhibits the system efficiency. For example, calling LLMs by prompts in the existing agent frameworks (e.g., Autogen, Langchain) under the concurrent setting predominantly employ a trial-and-error approach: prompts are fed into the LLM, converted to tensors, and loaded into GPU memory for execution. When CUDA memory capacity is exceeded, the system triggers an out-of-memory exception, deallocates the tensor, and signals failure to the requesting agent, necessitating repeated retry attempts until successful execution. This strategy significantly impacts system throughput and increases agent response latency, particularly in environments where multiple agents compete for limited GPU resources during inference.

To mitigate the limitations of deploying and running LLM-based agents, we introduce AIOS, an architecture designed to serve LLM-based agents more efficiently. Our contributions can be summarized in four parts.

○ ***New Agent-serving Architecture.*** We introduce AIOS, a novel architecture for serving LLM-based agents. This architecture divides agent applications and their accessible resources such as LLMs and tools into distinct layers, i.e., the application layer and the kernel layer. This separation enables more systematic resource management, efficiency optimization, and safety enhancement.

○ ***AIOS Kernel Design and Implementation.*** At the core of AIOS, we design and implement an AIOS kernel. In this kernel, agent primitives are designed to decompose LLM-related queries into sub execution units to enhance concurrency. To orchestrate the execution of these agent primitives, we develop an agent scheduler for scheduling and dispatching primitives to appropriate execution modules. Additionally, we implement memory, storage, and tool managers, along with the LLM core(s), to handle the execution of dispatched primitives. To prevent long-context requests consuming the LLM resource, we design a context manager to handle context interruptions and recoveries in the LLM core(s), especially in long-context scenarios. Moreover, an access manager is implemented to verify agent access rights before executing operations.

○ ***AIOS-Agent SDK Development.*** We develop the AIOS-Agent SDK, which provides a higher level abstraction of kernel functionalities, allowing developers to focus on application logic and higher-level functionalities without being burdened by the implementation details in the kernel.

○ ***Empirical Results.*** We conduct extensive evaluations of AIOS on agents developed using various agent frameworks. The experimental results demonstrate that AIOS can maintain the performance of agents across a wide range of standard benchmarks and can even enhance performance in benchmarks that involve calling external tools under the concurrent execution conditions. Furthermore, AIOS significantly improves execution efficiency, achieving up to a $2.1\times$ increase in execution speed for serving agents across different frameworks. These experimental results underscore the effectiveness of AIOS in optimizing both agent performance and execution speed in supporting diverse agent frameworks in resource-restricted environments.

2 The Architecture of AIOS

As depicted in Figure 2, the AIOS architecture is divided into three distinct layers: the application, kernel, and hardware layers. This layered design is intended to establish a clear separation of concerns within the system. Higher-level applications abstract the complexities of the underlying layers, interacting with them through well-defined interfaces such as software development kits (SDKs) and system calls.

Application Layer. At the application layer, agent applications are developed using the AIOS-Agent SDK, which provides the interface for requesting system resources through invoking system calls. On one hand, by using the SDK to request resources, agents are relieved from the burden of handling resource management. On the other hand, the SDK also facilitates isolation, ensuring that system resources cannot be directly manipulated by agents. The AIOS-Agent SDK is designed not only to support agents developed by using the native SDK functions, but also to facilitate the integration of non-native agents built with various agent creation frameworks, such as ReAct (Yao et al., 2023), Re-

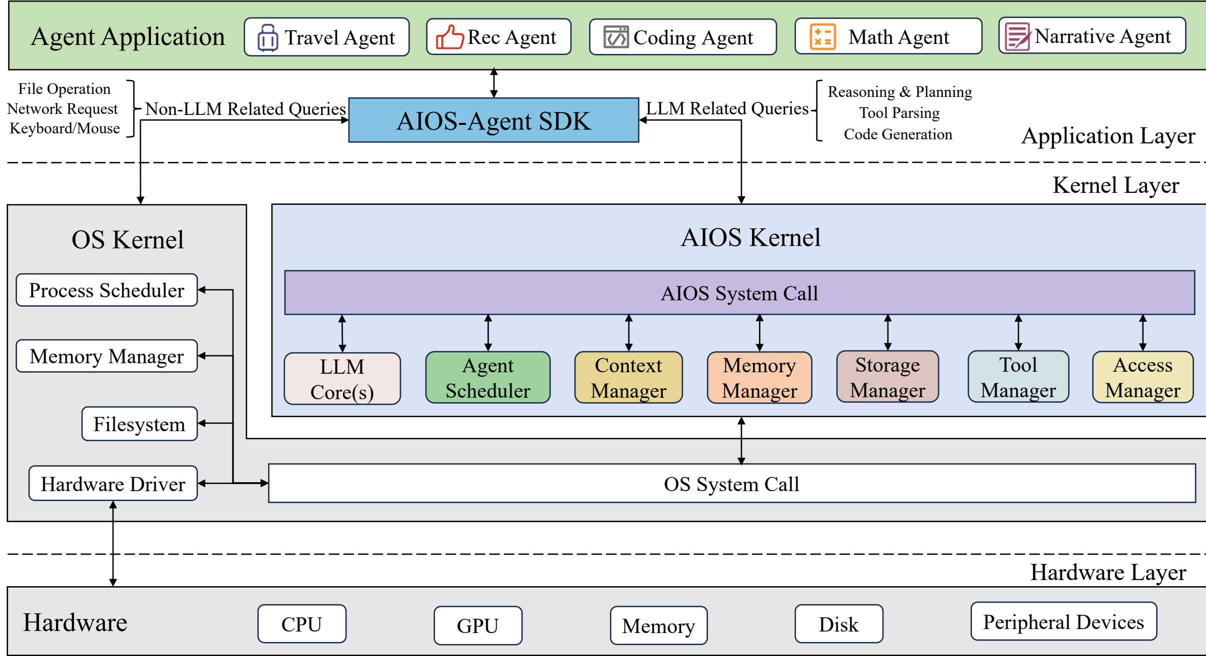


Figure 2. An overview of the AIOS architecture where responsibilities are isolated across different layers. Application layer facilitates the design and development of agent applications. Kernel layer manages core functionalities and resources to serve agent applications. Hardware layer controls and manages physical computing resources and devices to support kernel layer functionalities.

flexion (Shinn et al., 2023), Autogen (Wu et al., 2023), Open-Interpreter (Lucas, 2024), and MetaGPT (Hong et al., 2023). By providing the agent adapter function, the SDK supports non-native agents by allowing them to interact with the AIOS kernel resources. For native agent development, the SDK simplifies the creation of agents by offering predefined modules and APIs to achieve functionalities through invoking system calls, so that agents can request resources provided and managed by the AIOS kernel. This helps developers focus on the agent's primary workflows and logic rather than low-level implementation details.

Kernel Layer. The kernel layer is composed of two distinct yet synergistic components: the traditional OS kernel and the specialized AIOS kernel, each fulfilling unique roles within the system's functionality. The OS kernel retains its conventional architecture to manage non-LLM related computing tasks, while our core innovation centers around the AIOS kernel. Within the AIOS kernel, several modules are designed to facilitate agent requests through AIOS system calls. A scheduler is designed to dispatch these system calls to appropriate modules and employ strategies for scheduling AIOS system calls, which we will discuss in detail in Section 3.3. To facilitate the integration of diverse LLM endpoints, we design a unified interface that encapsulates LLMs as cores, akin to CPU cores, thereby allowing the integration of various LLM endpoints via a single interface. Additionally, to support context switching for LLMs, a context manager is introduced with mechanisms for context snapshot and restoration, further detailed in Section 3.4. To optimize agent memory handling, we develop a memory manager for managing agent memory operations and a storage manager for persistent storage operations, which will be explained further in Section 3.5 and Section 3.6, respectively. In addition, a tool manager is designed to load tools and manage tool call conflicts for the tools supported in the AIOS-Agent SDK, which will be covered in Section 3.7. Lastly, an access manager is designed with access control and user intervention, which we elaborate on in Section 3.8.

Hardware Layer. The hardware layer consists of the physical components of the system, such as the CPU, GPU, memory, disk, and peripheral devices. The hardware layer is not the main focus of the work—AIOS kernel does not directly interact with the hardware but relies on OS system calls to access the physical resources in the hardware layer.

3 AIOS KERNEL

In this section, we start with an overview of the AIOS kernel, highlighting how each module collaborates with other modules to support integrated functionalities. Following this, we provide an in-depth look into the design and implementation of each module, discussing their roles and contributions to the overall AIOS architecture.

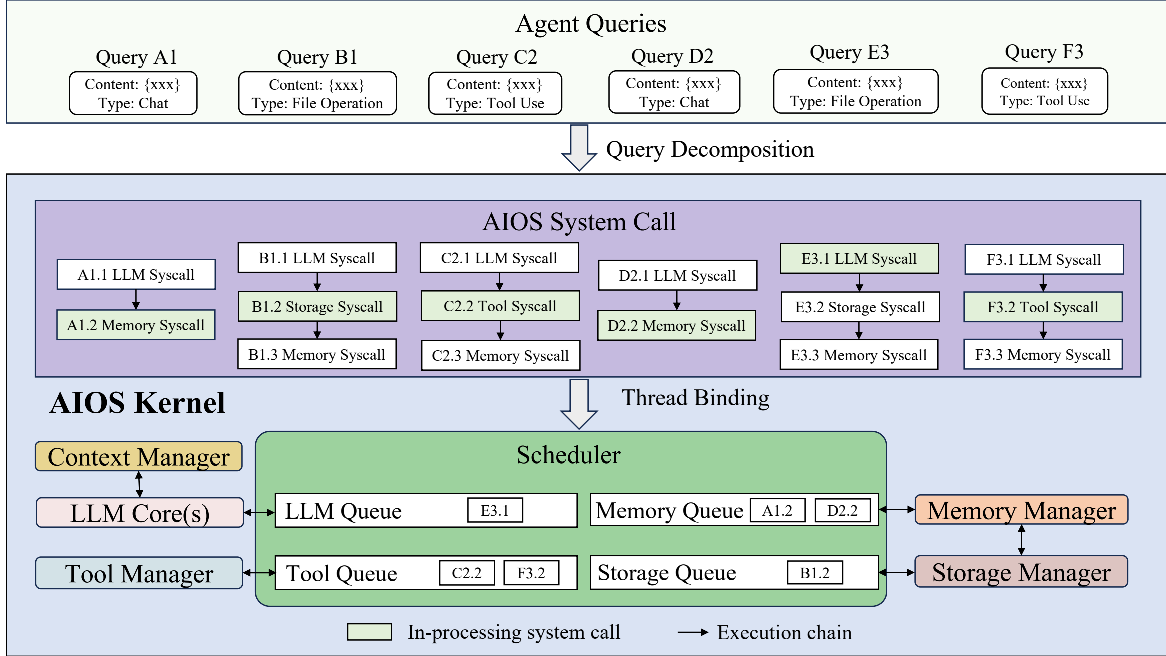


Figure 3. How agent queries are decomposed into AIOS system calls and how AIOS system calls are dispatched and scheduled. We omit the access manager module here as the access-related system calls will not be dispatched by the scheduler.

Table 1. AIOS system calls that are dispatched in the scheduler.

Module	AIOS System Call
LLM Core(s)	llm_generate
Memory Manager	mem_alloc, mem_read, mem_write, mem_clear
Storage Manager	sto_create, sto_read, sto_write, sto_clear, sto_retrieve
Tool Manager	tool_run

Table 2. Supported LLM instances in AIOS and the corresponding deployment options (no offline option for closed-source LLMs).

	Online	Offline
Open-source	Bedrock	Huggingface, vllm, Ollama
Closed-source	GPT, Claude, Gemini, Grok	-

3.1 Relationship and Connection between Modules

In the AIOS kernel, queries from agent applications are decomposed into distinct AIOS system calls, each categorized by functionality, such as LLM processing, memory access, storage operations, or tool usage, as illustrated in Figure 3. A subset of these system calls is shown in Table 1, while a comprehensive list can be found in Appendix A.1.

After decomposition, each system call is bound to an execution thread and subsequently dispatched by the scheduler. The scheduler centralizes and manages multiple queues for various modules, such as the LLM core(s), memory manager, storage manager, and tool manager. As a result, system calls are directed to the appropriate queue based on a specific attribute set assigned to each call. Each module listens to its corresponding queue in the scheduler and fetches the system calls scheduled to process. Among these processing modules, context manager is responsible for handling interruptions that may occur during the execution of system calls in the LLM core(s) (Section 3.4). Additionally, there is internal data swap between the memory manager and storage manager due to memory limitations. This modular architecture enables key components, such as the LLM core(s), memory manager, storage manager, and tool manager, to process requests concurrently within dedicated queues, enhancing isolation and parallelism. Thread binding implementations are detailed in Appendix A.1, while the data swap between memory and storage manager is covered in Section 3.5.

3.2 LLM Core(s)

Due to the various deployment options of LLMs, e.g., which LLM is used, whether the LLM is hosted on cloud or on local device, what hardware conditions the LLM requires, or which inference framework is used, we encapsulate each LLM instance adopting different deployment options as a core, akin to a CPU core in a traditional operating system.

This design allows us to treat each LLM instance as a dedicated processing unit, enhancing the modularity and extensibility within the AIOS architecture. To accommodate different LLM instances, we introduce a wrapper for each LLM instance and design unified system calls within this wrapper specifically for LLM inference. By abstracting an LLM instance as a core and implementing standardized system calls, AIOS provides a flexible way to integrate LLM

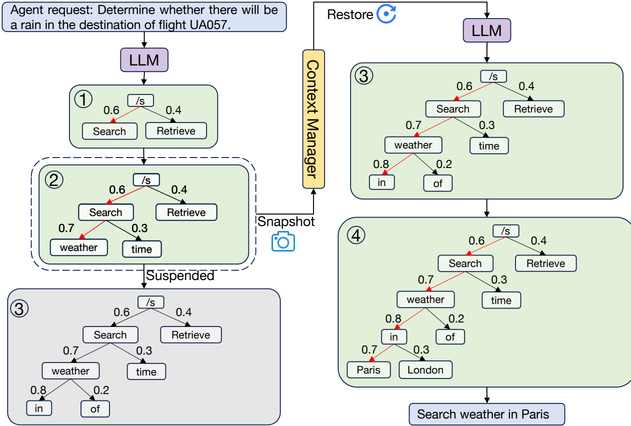


Figure 4. Illustration of the logits-based context snapshot and restoration process. We use beam search algorithm where beam width is set to 1 as an example.

instances under different deployment options, attributed to the modular design of the LLM core(s). Detailed information of LLM core(s) is provided in Appendix A.2.

3.3 Scheduler

Instead of placing separate queues within each processing module (e.g., LLM core(s) or memory manager), we centralize all the queues within the scheduler module. This approach isolates the responsibility for request management from the individual modules, allowing each processing module to focus on its execution. Besides, centralizing queue management in the scheduler simplifies the coordination of tasks across modules and provides a unified framework for scheduling. To schedule the AIOS system calls dispatched for each module in queues, we utilize two classic scheduling algorithms: First-In-First-Out (FIFO) and Round Robin (RR) due to their effectiveness and simplicity. The FIFO strategy processes system calls in the order they arrive, ensuring a straightforward handling sequence but potentially leading to increased waiting times for system calls queued later. In contrast, the RR strategy cycles through system calls in a time-sliced manner, allowing for more balanced resource distribution and reduced waiting times under high load conditions. To support time-slicing for the RR scheduling strategy, we introduce the context interrupt mechanism for LLM inference, which will be introduced in Section 3.4. Our centralized queue architecture provides a flexible foundation that accommodates diverse scheduling optimizations, from basic to sophisticated strategies. We provide the detailed implementation in Appendix A.3.

3.4 Context Manager

The inference time of LLMs is a critical bottleneck that can lead to long-running system calls, potentially monopolizing system resources. To address this issue and ensure efficient resource management, we design a context interrupt mechanism. This mechanism allows for the interruption and resumption of tasks through context snapshot and restoration operations, preventing prolonged system calls from dominating the LLM inference process.

The context manager designs two methods to capture and restore context based on different decoding strategies: text-based and logits-based approaches. For closed-source LLMs without logits access, the text-based approaches directly save the decoded text outputs and follow the previous decoding strategy at intermediate stages. Conversely, the logits-based approach preserves the structure of the intermediate search tree generated during inference, allowing for more fine-grained restoration of the computational state. This approach can be particularly advantageous for maintaining continuity in tasks requiring complex decoding strategy. The detailed procedure for the logits-based method is illustrated in Figure 4. We use the beam search process, a typical practice in LLMs (Touvron et al., 2023b; Jiang et al., 2023; Biderman et al., 2023), to illustrate the generative decoding process. For simplicity of illustration, we set the beam width as 1. Specifically, consider the prompt to the LLM as: *Determine whether there will be rain in the destination of flight UA057*. At each step, the LLM evaluates multiple candidate tokens, with the most promising paths kept for further expansion based on the predefined beam width. When the generation process is suspended by the scheduler at an intermediate step, the context manager uses the snapshot function to capture and store the current intermediate outputs of the LLM. Upon resumption, the restoration function is employed to reload the saved output from the snapshot, allowing the LLM to continue its generation process exactly from the point of suspension to reach the final answer: *Search weather in Paris*. In this way, the context manager ensures that the temporary suspension of one agent's request does not lead to a loss of progress, thereby improving efficiency since it does not need to generate from scratch.

3.5 Memory Manager

Unlike traditional OS memory manager that handles physical memory management such as RAM, the "memory" under the context of LLM-based agent refers to an agent's interaction history during the agent's runtime (Lerman & Galstyan, 2003; Zhang et al., 2024), such as the agent's conversation history with the LLM and the execution results of tool-calling. As a result, the memory manager in AIOS handles the management of these agent memories during the agent's runtime, such as memory structure, allocation, read, write, deletion, update, and compression. Agent memory is stored and managed on RAM by default, but when the agent's allocated RAM space is used up, the memory manager will swap agent's memory between RAM and disk through an eviction policy. More details are in the following.

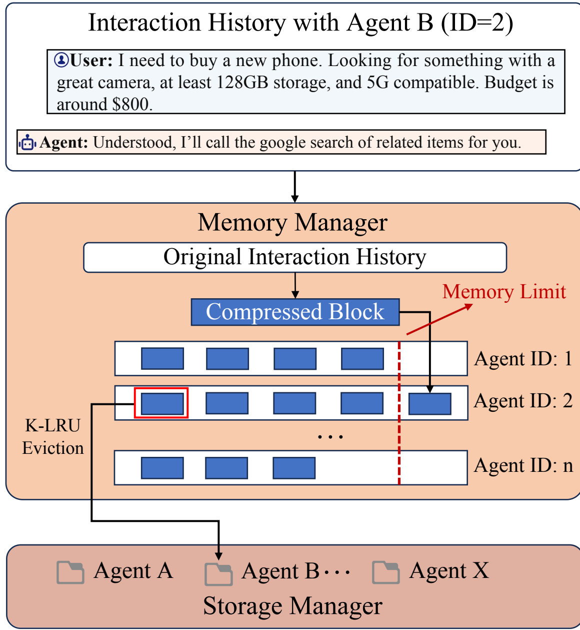


Figure 5. Illustration of memory and storage manager as well as their relationship. An agent's memory item in its memory block will be evicted to storage if its memory usage exceeds the memory limit, which is set to 80% of the memory block size. This threshold is configurable through AIOS configuration.

Trie-based Compression. The memory manager will allocate a fixed size memory block in RAM for each agent based on its runtime ID. This memory block is sized to remain within the LLM's maximum context length so that even agents construct prompts by combining all the memory items from its memory block, the prompts will not exceed the LLM's context window to cause crashes. To optimize memory utilization, the memory manager leverages a prefix-tree (trie) based compression mechanism when writing memories, since the memory items of an agent often contain overlapping prefixes. Implementation details of the memory compress are presented in Appendix A.5.

Memory Eviction Policy. As mentioned above, the memory block for each agent has a size limit, which is stored in RAM by default. If an agent's memory usage exceeds the limit of its memory block, e.g., 80% of its allocated memory block, the memory manager swaps agent's memory items between RAM and disk by initiating a K-Least Recently Used (K-LRU) eviction policy, which evicts memory items stored in RAM to disk by calling the storage manager (to be introduced in Section 3.6). The K-LRU eviction policy prioritizes keeping frequently accessed memory items in the RAM memory block, while less frequently used memory items will be moved by the storage manager to the disk. This approach balances memory efficiency and enables the system to offload less frequently visited data to disk and retrieves it when needed. The details of the K-LRU eviction policy are also presented in Appendix A.5.

3.6 Storage Manager

The storage manager handles persistent data storage for agents, such as files or knowledge bases that agents depend on to run and the agent memories that need to be persistently stored. The storage manager uses the same trie-based compression technique as the memory manager to optimize data storage. During an agent's runtime, when the agent's memory usage exceeds the allocated limit, the memory manager calls the storage manager to swap the data into the disk. Specifically, the storage manager reads and writes data based on the agent ID passed from the memory manager. In addition to the memory manager, the agent itself may also request to read and write data on disk during runtime, and these agent requests are also handled by the storage manager. Specifically, the agent calls the storage API in the SDK, which is further converted into storage-related system calls and put into the storage queue by the scheduler. The storage manager then processes the requests in the queue to fulfill the agent requests. The storage manager is implemented using local files and vector database (e.g., chromadb). Implementation details of the storage manager are included in Appendix A.6.

3.7 Tool Manager

The tool manager in the AIOS kernel is responsible for managing a broad suite of API tools supported by the AIOS-Agent SDK. When a tool is called by the tool name, the tool manager dynamically loads the associated tool instance by referencing its registration data stored within the system. This process ensures that the necessary implementation details, such as the tool's executable location, required libraries, and dependencies, are properly initialized. The tool manager utilizes a standardized interface to invoke the tool, allowing it to manage diverse tools under a uniform structure. Before executing a tool, the tool manager integrates a parameter validation process, verifying input parameters to prevent crashes of calling tools.

Resolution of Tool Call Conflicts. To manage tools with parallel access restrictions and usage limits, the tool manager uses a hashmap to track the number of instances of each tool currently being executed in the system. This hashmap helps monitor each tool's status and resolve conflicts arising from access limitations. When processing requests in the queue, the tool manager first checks the hashmap to determine whether the requested tool has reached its maximum usage limit or parallel access limit. If a conflict is detected, for example, when a tool has reached its parallel access limit, the manager moves to the next request in the queue and repeats this check. This process continues until the manager identifies a request that can be processed without conflicts. The implementation details are presented in Appendix A.7.

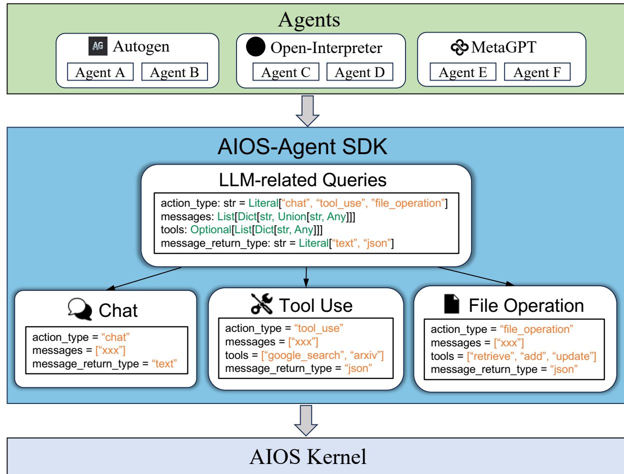


Figure 6. Illustration of how agent applications leverage the AIOS-Agent SDK to send queries to the AIOS Kernel. For simplicity, queries sent directly to the OS kernel are omitted.

3.8 Access Manager

The access manager in the AIOS kernel provides the following two key functionalities.

Access Control. The access manager controls read and write permissions when some agents request read and write operation of other agent's data such as memory. The access manager achieves this by assigning each agent to a specific privilege group. Agents can only access resources, such as LLM interaction history or tool usage logs of other agents only if they are within other agents' privilege group. To manage these permissions, the access manager uses a hashmap, where each agent's ID is mapped to its assigned privilege group. When an agent requests access to a resource, the access manager checks the hashmap to verify the agent's permissions before allowing the request to proceed.

User Intervention. To prevent accidental or unintended operations when users interact with AIOS, particularly irreversible operations such as deletion, overwrite and privilege change, we provide a user intervention interface. This interface provides users with prompt checks to confirm before these irreversible operations are executed over files or tools. The implementation details are presented in Appendix A.8.

3.9 AIOS-Agent SDK

We design the AIOS-Agent SDK to streamline the development and integration of agents on the AIOS architecture. This SDK not only empowers developers to build agents that interact with the core functions in the AIOS kernel but also abstracts complex system calls, allowing developers to focus on the agent's internal workflows.

Tool Integration. To support diverse agent functionalities, the AIOS-Agent SDK integrates a wide range of tools sourced from various platforms and supports them natively, covering both online and offline capabilities and supporting multiple input-output modalities. This integration allows agents to access resources and tools seamlessly, regardless of their origins or configurations. Detailed information on these integrated tools is provided in Appendix B.2.

Interaction Interface with the AIOS Kernel. To facilitate the utilization of functions provided by AIOS system calls in the AIOS kernel, the SDK defines three main functions that agents can use to invoke system calls and request resources. These functions categorize agent queries based on the type of action, streamlining operations and improving resource management. As shown in Figure 6, the SDK defines three main types of agent queries: *chat*, *tool_use*, and *file_operation*, each specified by the *action_type* attribute. Through the functions provided by SDK, agents can communicate with the AIOS kernel to access resources by invoking system calls instead of directly manipulating resources.

Agent Framework Adapter. To support agents built with various agent creation frameworks, such as Autogen (Wu et al., 2023), Open-Interpreter (Lucas, 2024), and MetaGPT (Hong et al., 2023), the AIOS-Agent SDK provides adapters for these frameworks. These adapters locate the core functions in the aforementioned frameworks and redirect them to the functions in AIOS. This adaptation allows agents from different frameworks to operate within the AIOS environment without modification of the agent code. Further details on the core functions and specific adaptations for each agent framework are provided in Appendix B.4.

4 Evaluation

In this section, we conduct experiments to answer the following research questions.

• RQ1: Can AIOS maintain or even enhance the performance of agents on standard benchmarks when running multiple agent instances simultaneously?

• RQ2: How effectively can AIOS optimize system execution throughput and reduce response latency when serving numerous agents built with different agent frameworks?

• RQ3: How scalable is AIOS as the number of concurrently running agents increases?

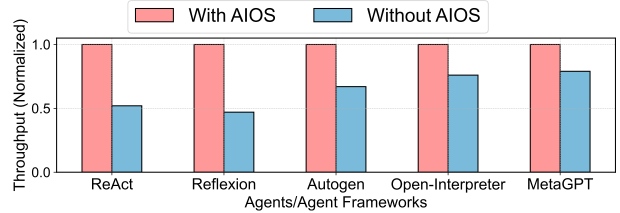
4.1 Setup

Models. We use the GPT-4o-mini (Achiam et al., 2023) as the closed-source API, and use two open-source LLMs, i.e., Llama-3.1-8b (Dubey et al., 2024) and Mistral-7b (Jiang et al., 2023), as the LLM core, respectively, during the experiments. The open-source models are both instruction-tuned versions and we use float16 precision.

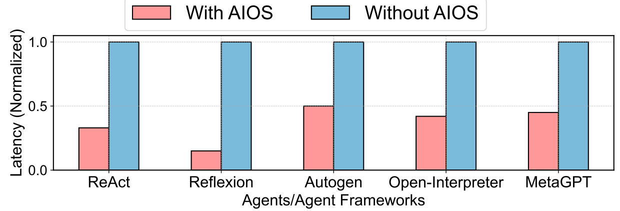
Hardware. Our experiments are conducted on an Ubuntu 22.04 machine equipped with NVIDIA RTX A5000 GPUs (24GB). We run all experiments using a single A5000 GPU.

Table 3. Evaluation of agent performance on benchmarks w/o and w/ AIOS, respectively. Success rate (SR%) is used as the metric for all the benchmarks. "-" represents methods that failed GAIA benchmark tasks due to lack of API support.

Method	HumanEval	MINT (Code)	GAIA	SWE-Bench-Lite
ReAct w/o AIOS	48.8	29.4	5.5	3.9
ReAct w/ AIOS	**50.6**	**30.1**	**7.3**	**4.3**
Reflexion w/o AIOS	50.6	32.4	6.7	4.7
Reflexion w/ AIOS	**51.8**	**33.8**	**7.8**	**5.1**
Autogen w/o AIOS	87.8	42.5	7.3	4.3
Autogen w/ AIOS	87.8	42.5	**9.7**	4.3
Open-Interpreter w/o AIOS	85.4	45.9	-	4.7
Open-Interpreter w/ AIOS	**86.0**	**48.7**	-	**5.1**
MetaGPT w/o AIOS	82.9	41.1	-	5.9
MetaGPT w/ AIOS	82.9	**41.8**	-	5.9



(a) Normalized throughput. Higher is better.



(b) Normalized latency. Lower is better.

Figure 7. Efficiency analysis on different agent frameworks evaluated on the Llama-3.1-8b model on the HumanEval benchmark.

Agent Frameworks. We conduct evaluation by running agents built from various popular agent frameworks: ReAct (Yao et al., 2023), Reflexion (Shinn et al., 2023), Autogen (Wu et al., 2023), Open-Interpreter (Lucas, 2024) and MetaGPT (Hong et al., 2023). Details of these agent frameworks are introduced in Appendix B.4.

Workloads. We evaluate on a resource-constrained scenario in which agents run concurrently with a single LLM deployed that can process only one prompt request at a time. To create these concurrent conditions, we set the maximum number of working threads to 250 by default, i.e., at most 250 agents can run concurrently at the same time. The impact of increasing the number of agents will be analyzed in Section 4.4. By default, we use RR as the scheduling strategy for AIOS to run agents. The impact of using other strategy (i.e., FIFO) is reported in Section 4.3.

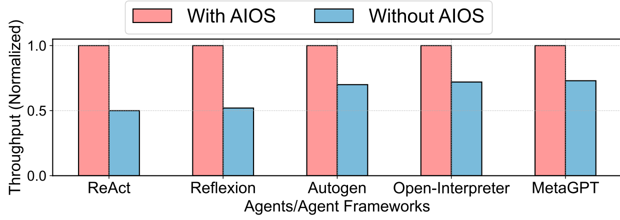
4.2 Agent Performance (RQ1)

To evaluate whether using AIOS can maintain or even improve the agent performance on standard benchmarks, we adopt four agent benchmarks, i.e., HumanEval (Chen et al., 2021a), MINT (the code subset) (Wang et al., 2023b), GAIA (Mialon et al., 2023) and SWE-Bench-Lite (Jimenez et al., 2024) to run agents without and with AIOS, respectively. We use the success rate (SR%) as the metric, consistent with the original benchmarks and use GPT-4o-mini as the LLM core to run all the agents. To eliminate randomness, we set the temperature to 0 for GPT-4o-mini in all experiments. Detailed descriptions of the benchmark setups and configurations can be found in Appendix C.

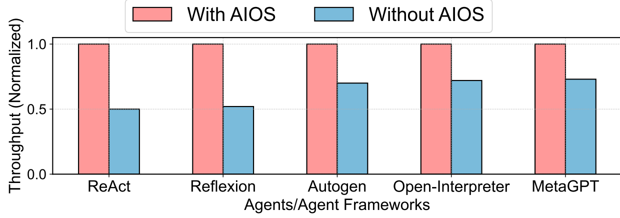
As shown in Table 3, incorporating AIOS consistently maintains agent performance across standard benchmarks. In some cases, AIOS can also contribute to agent performance improvements. For example, in code generation benchmarks such as MINT, HumanEval, and SWE-Bench-Lite, AIOS boosts agent performance by prompt enhancement, which embeds the system prompts with more structural input and output within the LLM wrapper. These enhanced prompts provide the LLM with additional context and structural guidance for higher-quality code generation. In tool calling benchmarks like GAIA, agent performance is boosted for two main reasons. First, the tool manager implements a post-verification process using structural regex to ensure that the input parameters for tool calls conform the correct format. This extra validation step helps prevent errors by catching incorrect tool names or parameters generated by the LLM before the tool call is executed. Second, AIOS employs conflict resolution to manage tool calls, preventing conflicts that might otherwise cause successful tool calls to fail. By mitigating issues from concurrent tool access, AIOS ensures stable operation for agents during execution.

4.3 Efficiency Analysis (RQ2)

In our efficiency experiments, we evaluate system performance using two key metrics: **throughput** and **latency**. Throughput is measured by counting the number of AIOS system calls executed per second, indicating the system's capacity to handle multiple requests in parallel. Latency, on the other hand, is measured as the average waiting time experienced by agents, from the moment a query is submitted to the completion of the response, reflecting the system's responsiveness. To ensure a controlled and consistent testing environment, we conduct these evaluations using the two open-source models, Llama-3.1-8b and Mistral-7b, both hosted locally. Hosting these models locally reduces potential variability in LLM API response times due to network-related latency issues. As shown in Figure 7a and Figure 8a, the results demonstrate that AIOS achieves significantly higher throughput across different agent frameworks, to a $2.1\times$ increase in throughput when using Reflexion-based



(a) Normalized throughput. Higher is better.



(b) Normalized latency. Lower is better.

Figure 8. Efficiency analysis on different agent frameworks evaluated on the Mistral-7b model on the HumanEval benchmark.

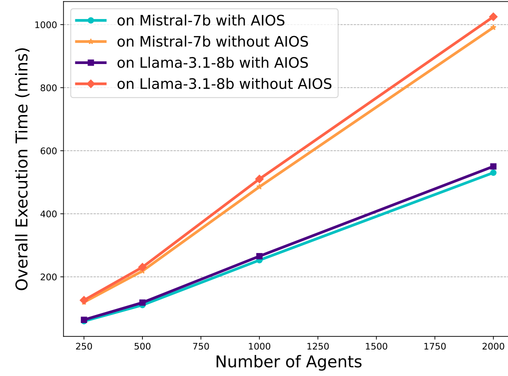
Table 4. Impact of using different scheduling strategies, where NONE represents without using AIOS, FIFO and RR represent using AIOS with the two different scheduling strategies. All metrics are reported in minutes, including overall execution time and agent waiting time (average and p90).

Strategy	Overall execution time	Agent waiting time	
		Avg.	p90
None	152.1	9.8	11.0
FIFO	**74.2**	**3.0**	5.0
RR	77.3	3.2	**4.2**

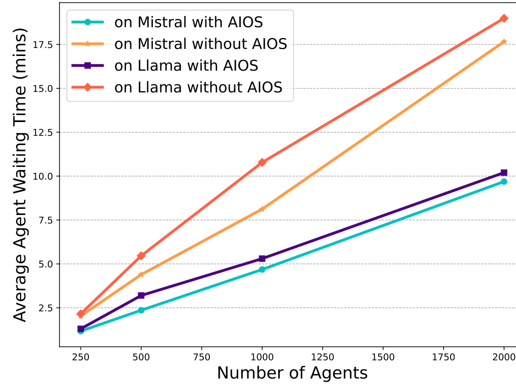
agents on Llama-3.1-8b. This improvement is attributed to the scheduling employed in the AIOS kernel, which prevents unnecessary trial-and-error attempts by avoiding prompts that cannot be loaded onto the GPU for execution. In terms of latency, as illustrated in Figure 7b and Figure 8b, the average waiting time for agents is also substantially reduced. This reduction highlights the efficiency of AIOS in serving LLM-based agents.

Impact of Different Scheduling Strategies. To further analyze the impact of different scheduling strategies on system efficiency, we conduct an ablation study using agents built with ReAct on the HumanEval benchmark with the Llama-3.1-8b model. We test three strategies: without AIOS, FIFO, and Round Robin (RR), and measure the overall execution time and agent waiting time (average and p90).

As shown in Table 4, the FIFO strategy achieves the shortest overall execution time compared to the other strategies. RR comes second in terms of overall execution and average agent waiting time, as its context switching introduces additional overhead. However, RR performs better on the p90 metric (i.e., the value below which 90% of waiting times fall) due to its fairer scheduling approach, which reduces the likelihood of later tasks having longer waiting time, which can typically occur in FIFO.



(a) Overall Execution Time v.s. Agent Number.



(b) Average Agent Waiting Time v.s. Agent Number.

Figure 9. Overall execution time and average agent waiting time when agent number increases from 250 to 2000.

4.4 Scalability Analysis (RQ3)

In this section, we evaluate the scalability of AIOS by progressively increasing the number of active agents from 250 to 2000. These experiments were conducted using the Llama-3.1-8b and Mistral-7b models on the HumanEval benchmark. Since the HumanEval dataset contains only 164 samples, we scaled up the dataset by duplicating samples to match the increasing number of agents, enabling large-scale concurrent execution of agent instances. As shown in Figure 9a and Figure 9b, the results demonstrate that using AIOS can allow both the overall execution time and the average agent waiting time to maintain an approximate linear relationship with the number of agents. This predictable, linear scaling illustrates AIOS's ability to handle increasing workloads efficiently, even as demand intensifies. In contrast, without AIOS, the execution and waiting times increase at a faster rate. The gap between using AIOS and

not using AIOS widens as the number of agents increases, underscoring AIOS's effectiveness in managing concurrent operations. As workloads scale, AIOS can still maintain system stability and responsiveness, reducing both execution and waiting times compared to configurations without AIOS. This growing performance advantage highlights AIOS's suitability for environments with high or fluctuating workloads, demonstrating its potential to serve a large number of agents.

5 Related Work

5.1 Evolution of Operating Systems

The evolution of operating systems (OS) has unfolded in a progressive way, evolving from rudimentary systems to the complex and interactive OS of today. Their evolution saw a transition from simple batch job processing (IBM, 2010) to more advanced process management techniques like time-sharing (Ritchie & Thompson, 1974) and multi-task processing (Hoare, 1974; Engler et al., 1995), which facilitated the handling of increasingly complex tasks. The progress moved toward modularization within the OS, delineating specific responsibilities such as process scheduling (Liu & Layland, 1973; Dijkstra, 2002), memory management (Denning, 1968; Daley & Dennis, 1968), and filesystem management (Rosenblum & Ousterhout, 1992; McKusick et al., 1984), enhancing efficiency and manageability. The further advent of graphical user interfaces (GUIs), e.g., Macintosh, Windows and GNOME , makes operating systems more interactive and user-centric. Meanwhile, the operating system ecosystem has also expanded, offering a comprehensive suite of developer tools (OS SDKs) and runtime libraries. These tools enable application developers to design, implement, and run their applications efficiently within the OS environment (Ge et al., 2023b). Notable examples of OS ecosystems include Android Studio , XCode and Cloud SDK . In these ecosystems, the OS provides numerous resources to facilitate software development and serves as a platform for deploying and hosting software applications, leading to a thriving OS-application ecosystem. Recently, the community is seeing AI models such as LLMs sinking from the application layer down to the system layer to provide standard services to various applications. With the incorporation of large language models (LLMs), these advanced systems promise to further narrow the communication gap between humans and machines, forwarding a new era of user-computer interaction.

5.2 Large Language Model Agents

LLM-based single-agent systems (SAS) use a single LLM agent for complex task solving, such as travel planning (Xie et al., 2024), personalized recommendation, and artistic design (Ge et al., 2023a). The agent takes natural language instruction from users as input and decomposes the task into a multistep plan for task solving, where each step may call external tools to be completed, such as collecting information, executing specialized models, or interacting with the external world. Single-agent applications may engage with either digital environment or physical environment or both, depending on the task to solve. For example, agents in virtual or digital environment may invoke APIs (Ge et al., 2023a; Schick et al., 2023; Yao & Narasimhan, 2023; Parisi et al., 2022; Tang et al., 2023; Xie et al., 2024), browse websites (Nakano et al., 2022; Deng et al., 2023; Wu et al., 2024), or execute codes (Zhang et al., 2023; Yang et al.), while agents in the physical environment may manipulate objects (Brohan et al., 2023; Fan et al., 2022; Wang et al., 2023a), carry out lab experiments (Boiko et al., 2023; Bran et al., 2023), or make actionable decisions (Huang et al., 2022; Xiang et al., 2023). LLM-based multi-agent systems (MAS) leverage the interaction among multiple agents for problem solving. The relationship among the multiple agents could be cooperative (Wang et al., 2023c; Mandi et al., 2023), competitive (Chan et al., 2023; Du et al., 2023), or a mixture of cooperation and competition (Ge et al., 2023b). In cooperative multi-agent systems, each agent takes and assesses the information provided by other agents, thereby working together to solve complex tasks, such as role playing (Li et al., 2023; Chen et al., 2023; Zhu et al., 2023), social simulation (Park et al., 2023) and software development (Hong et al., 2023; Qian et al., 2023; Wu et al., 2023; Josifoski et al., 2023). In competitive multi-agent systems, agents may debate, negotiate and compete with each other in a game environment to achieve their goals, such as improving negotiation skills (Fu et al., 2023) and debating about the correct answer (Du et al., 2023; Chan et al., 2023; Liang et al., 2023; Hua et al., 2023).

6 Conclusion and Future Work

This paper introduces AIOS, a novel architecture designed to serve LLM-based agents. Within this architecture, we design and implement an AIOS kernel that isolates resources and LLM-specific services from agent applications for management. Additionally, we develop the AIOS-Agent SDK to facilitate the usage of the functionalities provided by the AIOS kernel for agent applications. Experimental results demonstrate that AIOS not only maintains, but can also improve agent performance on standard benchmarks. Furthermore, AIOS significantly accelerates overall execution time, improves system throughput, and exhibits scalability as the number of concurrent agents increases. We hope that the insights and methodologies shared in this work will contribute to both AI and systems research, fostering a more cohesive, effective, and efficient ecosystem for serving LLM-based agents. We believe future research can explore innovative directions built upon AIOS to refine and expand AIOS architecture to better meet the evolving requirements of developing and deploying LLM-based AI agents.

REFERENCES

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Biderman, S., Schoelkopf, H., Anthony, Q. G., Bradley, H., O'Brien, K., Hallahan, E., Khan, M. A., Purohit, S., Prashanth, U. S., Raff, E., et al. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*, pp. 2397–2430. PMLR, 2023.

Boiko, D. A., MacKnight, R., and Gomes, G. Emergent autonomous scientific research capabilities of large language models. *arXiv preprint arXiv:2304.05332*, 2023.

Bran, A. M., Cox, S., White, A. D., and Schwaller, P. Chemcrow: Augmenting large-language models with chemistry tools. *arXiv preprint arXiv:2304.05376*, 2023.

Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., and Mylopoulos, J. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8:203–236, 2004.

Brohan, A., Chebotar, Y., Finn, C., Hausman, K., Herzog, A., Ho, D., Ibarz, J., Irpan, A., Jang, E., Julian, R., et al. Do as i can, not as i say: Grounding language in robotic affordances. In *Conference on robot learning*, pp. 287–318. PMLR, 2023.

Chan, C.-M., Chen, W., Su, Y., Yu, J., Xue, W., Zhang, S., Fu, J., and Liu, Z. Chateval: Towards better llm-based evaluators through multi-agent debate. In *The Twelfth International Conference on Learning Representations*, 2023.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code. 2021a.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021b.

Chen, W., Su, Y., Zuo, J., Yang, C., Yuan, C., Qian, C., Chan, C.-M., Qin, Y., Lu, Y., Xie, R., et al. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors in agents. *arXiv preprint arXiv:2308.10848*, 2023.

Chung, H. W., Hou, L., Longpre, S., Zoph, B., Tay, Y., Fedus, W., Li, Y., Wang, X., Dehghani, M., Brahma, S., et al. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*, 2022.

Daley, R. C. and Dennis, J. B. Virtual memory, processes, and sharing in multics. *Communications of the ACM*, 11 (5):306–312, 1968.

Deng, X., Gu, Y., Zheng, B., Chen, S., Stevens, S., Wang, B., Sun, H., and Su, Y. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36, 2023.

Denning, P. J. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

Dijkstra, E. W. Cooperating sequential processes. In *The origin of concurrent programming: from semaphores to remote procedure calls*, pp. 65–138. Springer, 2002.

Driess, D., Xia, F., Sajjadi, M. S., Lynch, C., Chowdhery, A., Ichter, B., Wahid, A., Tompson, J., Vuong, Q., Yu, T., et al. Palm-e: an embodied multimodal language model. In *Proceedings of the 40th International Conference on Machine Learning*, pp. 8469–8488, 2023.

Du, Y., Li, S., Torralba, A., Tenenbaum, J. B., and Mordatch, I. Improving factuality and reasoning in language models through multiagent debate. *arXiv preprint arXiv:2305.14325*, 2023.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Engler, D. R., Kaashoek, M. F., and O'Toole Jr, J. Exokernel: An operating system architecture for application-level resource management. *ACM SIGOPS Operating Systems Review*, 29(5):251–266, 1995.

Fan, L., Wang, G., Jiang, Y., Mandlekar, A., Yang, Y., Zhu, H., Tang, A., Huang, D.-A., Zhu, Y., and Anandkumar, A. Minedojo: Building open-ended embodied agents with internet-scale knowledge. *Advances in Neural Information Processing Systems*, 35:18343–18362, 2022.

Fu, Y., Peng, H., Khot, T., and Lapata, M. Improving language model negotiation with self-play and in-

context learning from ai feedback. *arXiv preprint arXiv:2305.10142*, 2023.

Ge, Y., Hua, W., Mei, K., Tan, J., Xu, S., Li, Z., and Zhang, Y. OpenAGI: When LLM Meets Domain Experts. *Advances in Neural Information Processing Systems*, 36, 2023a.

Ge, Y., Ren, Y., Hua, W., Xu, S., Tan, J., and Zhang, Y. LLM as OS, Agents as Apps: Envisioning AIOS, Agents and the AIOS-Agent Ecosystem. *arXiv:2312.03815*, 2023b.

Geng, S., Liu, S., Fu, Z., Ge, Y., and Zhang, Y. Recommendation as language processing (rlp): A unified pretrain, personalized prompt & predict paradigm (p5). In *Proceedings of the 16th ACM Conference on Recommender Systems*, pp. 299–315, 2022.

Hao, S., Gu, Y., Ma, H., Hong, J., Wang, Z., Wang, D., and Hu, Z. Reasoning with language model is planning with world model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 8154–8173, 2023.

Hoare, C. A. R. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Wang, J., Zhang, C., Wang, Z., Yau, S. K. S., Lin, Z., et al. Metagpt: Meta programming for multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2023.

Hua, W., Fan, L., Li, L., Mei, K., Ji, J., Ge, Y., Hemphill, L., and Zhang, Y. War and peace (waragent): Large language model-based multi-agent simulation of world wars. *arXiv preprint arXiv:2311.17227*, 2023.

Huang, W., Abbeel, P., Pathak, D., and Mordatch, I. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning*, pp. 9118–9147. PMLR, 2022.

IBM, C. What is batch processing? *z/OS Concepts*, 2010.

Jennings, N. R., Sycara, K., and Wooldridge, M. A roadmap of agent research and development. *Autonomous agents and multi-agent systems*, 1:7–38, 1998.

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., Casas, D. d. l., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. R. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024.

Josifoski, M., Klein, L., Peyrard, M., Li, Y., Geng, S., Schnitzler, J. P., Yao, Y., Wei, J., Paul, D., and West, R. Flows: Building blocks of reasoning and collaborating ai. *arXiv preprint arXiv:2308.01285*, 2023.

Kim, G., Baldi, P., and McAleer, S. Language models can solve computer tasks. *Advances in Neural Information Processing Systems*, 36, 2023.

Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., and Iwasawa, Y. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35: 22199–22213, 2022.

Lerman, K. and Galstyan, A. Agent memory and adaptation in multi-agent systems. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pp. 797–803, 2003.

Li, G., Hammoud, H., Itani, H., Khizbullin, D., and Ghanem, B. Camel: Communicative agents for "mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36, 2023.

Liang, T., He, Z., Jiao, W., Wang, X., Wang, Y., Wang, R., Yang, Y., Tu, Z., and Shi, S. Encouraging divergent thinking in large language models through multi-agent debate. *arXiv preprint arXiv:2305.19118*, 2023.

Liu, C. L. and Layland, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

Lucas, K. Open interpreter. https://github.com/OpenInterpreter/open-interpreter, 2024.

Mandi, Z., Jain, S., and Song, S. Roco: Dialectic multi-robot collaboration with large language models. *arXiv preprint arXiv:2307.04738*, 2023.

McKusick, M. K., Joy, W. N., Leffler, S. J., and Fabry, R. S. A fast file system for unix. *ACM Transactions on Computer Systems (TOCS)*, 2(3):181–197, 1984.

Mialon, G., Fourrier, C., Swift, C., Wolf, T., LeCun, Y., and Scialom, T. Gaia: a benchmark for general ai assistants. *arXiv preprint arXiv:2311.12983*, 2023.

Nakano, R., Hilton, J., Balaji, S., Wu, J., Ouyang, L., Kim, C., Hesse, C., Jain, S., Kosaraju, V., Saunders, W., Jiang, X., Cobbe, K., Eloundou, T., Krueger, G., Button, K., Knight, M., Chess, B., and Schulman, J. Webgpt: Browser-assisted question-answering with human feedback, 2022.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. Codegen: An open large

language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pp. 311–318, 2002.

Parisi, A., Zhao, Y., and Fiedel, N. Talm: Tool augmented language models. *arXiv preprint arXiv:2205.12255*, 2022.

Park, J. S., O'Brien, J., Cai, C. J., Morris, M. R., Liang, P., and Bernstein, M. S. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, pp. 1–22, 2023.

Qian, C., Cong, X., Yang, C., Chen, W., Su, Y., Xu, J., Liu, Z., and Sun, M. Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2023.

Qin, Y., Liang, S., Ye, Y., Zhu, K., Yan, L., Lu, Y., Lin, Y., Cong, X., Tang, X., Qian, B., et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. *ICLR*, 2024.

Ritchie, D. M. and Thompson, K. The unix time-sharing system. *Communications of the ACM*, 17(7):365–375, 1974.

Rosenblum, M. and Ousterhout, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

Ross, S. I., Martinez, F., Houde, S., Muller, M., and Weisz, J. D. The programmer's assistant: Conversational interaction with a large language model for software development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces*, pp. 491–514, 2023.

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., and Scialom, T. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2023.

Tang, Q., Deng, Z., Lin, H., Han, X., Liang, Q., and Sun, L. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *arXiv preprint arXiv:2306.05301*, 2023.

Taylor, R., Kardas, M., Cucurull, G., Scialom, T., Hartshorn, A., Saravia, E., Poulton, A., Kerkez, V., and Stojnic, R. Galactica: A large language model for science. *arXiv preprint arXiv:2211.09085*, 2022.

Team, G., Anil, R., Borgeaud, S., Wu, Y., Alayrac, J.-B., Yu, J., Soricut, R., Schalkwyk, J., Dai, A. M., Hauth, A., et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023a.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023b.

Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. Voyager: An open-ended embodied agent with large language models. In *Intrinsically-Motivated and Open-Ended Learning Workshop@ NeurIPS2023*, 2023a.

Wang, X., Wang, Z., Liu, J., Chen, Y., Yuan, L., Peng, H., and Ji, H. Mint: Evaluating llms in multi-turn interaction with tools and language feedback. *arXiv preprint arXiv:2309.10691*, 2023b.

Wang, Z., Mao, S., Wu, W., Ge, T., Wei, F., and Ji, H. Unleashing cognitive synergy in large language models: A task-solving agent through multi-persona self-collaboration. *arXiv preprint arXiv:2307.05300*, 1(2): 3, 2023c.

Wooldridge, M. and Jennings, N. R. Intelligent agents: Theory and practice. *The knowledge engineering review*, 10(2):115–152, 1995.

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Zhang, S., Zhu, E., Li, B., Jiang, L., Zhang, X., and Wang, C. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 2023.

Wu, Z., Han, C., Ding, Z., Weng, Z., Liu, Z., Yao, S., Yu, T., and Kong, L. Os-copilot: Towards generalist computer agents with self-improvement. *arXiv preprint arXiv:2402.07456*, 2024.

Xiang, J., Tao, T., Gu, Y., Shu, T., Wang, Z., Yang, Z., and Hu, Z. Language models meet world models: Embod-

ied experiences enhance language models. *Advances in neural information processing systems*, 36, 2023.

Xie, J., Zhang, K., Chen, J., Zhu, T., Lou, R., Tian, Y., Xiao, Y., and Su, Y. Travelplanner: A benchmark for real-world planning with language agents. *arXiv preprint arXiv:2402.01622*, 2024.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. Swe-agent: Agent-computer interfaces enable automated software engineering.

Yao, S. and Narasimhan, K. Language agents in the digital world: Opportunities and risks. *princeton-nlp.github.io*, 2023.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. ReAct: Synergizing reasoning and acting in language models. *International Conference on Learning Representations*, 2023.

Zhang, K., Li, G., Li, J., Li, Z., and Jin, Z. Toolcoder: Teach code generation models to use apis with search tools. *arXiv preprint arXiv:2305.04032*, 2023.

Zhang, T., Kishore, V., Wu, F., Weinberger, K. Q., and Artzi, Y. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*, 2019.

Zhang, Z., Bo, X., Ma, C., Li, R., Chen, X., Dai, Q., Zhu, J., Dong, Z., and Wen, J.-R. A survey on the memory mechanism of large language model based agents. *arXiv preprint arXiv:2404.13501*, 2024.

Zhu, X., Chen, Y., Tian, H., Tao, C., Su, W., Yang, C., Huang, G., Li, B., Lu, L., Wang, X., et al. Ghost in the minecraft: Generally capable agents for open-world viroments via large language models with text-based knowledge and memory. *arXiv preprint arXiv:2305.17144*, 2023.

APPENDIX

This appendix contains additional details for the paper: ***"AIOS: LLM Agent Operating System"***. The appendix is organized as follows:

- Section §A provides **AIOS Kernel Implementation Details**.
- Section §B reports more about **AIOS-Agent SDK**.
- Section §C reports more **Details of Agent Benchmarks**.
- Section §D shows more **Additional Experimental Results**.
- Section §E analyzes **Discussion**.

A AIOS KERNEL IMPLEMENTATION DETAILS

A.1 AIOS System Call

The modules in AIOS achieve their functionalities by invoking system calls. Table 5 shows a more comprehensive list of system calls correspondent to different modules and present the arguments for invoking these system calls.

Table 5. AIOS System Calls.

Module	AIOS System Call	Arguments
Scheduler	set_id	aid: int
	get_id	-
	set_status	status: string
	get_status	-
	set_priority	priority: int
	get_priority	-
LLM core(s)	llm_generate	prompt: list
Memory Manager	mem_alloc	aid: int
	mem_read	aid: int, rid: int
	mem_write	aid: int, rid: int, s: string
	mem_clear	aid: int, rid: int
Storage Manager	sto_create	aname: string, aid: int, rid: int
	sto_read	aname: string, aid: int, rid: int
	sto_write	aname: string, aid: int, rid: int, s: string
	sto_retrieve	aname: string, aid: int, rid: int, query: string
	sto_clear	aname: string, aid: int, rid: int
Tool Manager	tool_run	params: dict
Context Manager	gen_snapshot	cid: int, data: bytes | string
	gen_restore	cid: int
	check_restore	cid: int
	clear_restore	cid: int
Access Manager	check_access	sid: int, tid: int
	ask_permission	aid: int, operation: string

Thread Binding. Each system call within AIOS is bound to a separate thread for execution, allowing for concurrent processing. The thread binding is implemented by inheriting the **Thread** class and overwrites its init and run methods.

```python
class SysCall(Thread):
    def __init__(self, agent_name, request_data):
        super().__init__()
        self.agent_name = agent_name
        self.request_data = request_data
        self.event = threading.Event()
        self.pid = None
        self.status = None
        self.response = None
        self.time_limit = None
        self.created_time = None
        self.start_time = None
        self.end_time = None

    def run(self):
        self.set_pid(self.native_id)
        self.event.wait()
```

A.2 LLM Core(s)

AIOS provides multiple different LLM instances for agents to choose, and an agent can choose one LLM instance as the core during runtime. The structure for implementing different LLM cores is defined in the **LLMCore** abstract class, which defines the unified function interface for different LLM instances.

```
class LLMCore(ABC):
    def __init__(self,
                 llm_name: str,
                 max_gpu_memory: dict = None,
                 eval_device: str = None,
                 max_new_tokens: int = 256,
                 log_mode: str = "console"):
        """  Initialize LLMCore with model configurations
        """
        pass

    @abstractmethod
    def load_llm_and_tokenizer(self) -> None:
        """  Load the LLM model and tokenizer
        """
        pass

    def tool_calling_input_format(self,
                                  prompt: list,
                                  tools: list) -> list:
        """  Format prompts to include tool information
        """
        pass

    def parse_tool_calls(self, tool_call_str):
        """  Parse and add tool call identifiers for models without tool support
        """
        pass

    @abstractmethod
    def address_request(self,
                        llm_request,
                        temperature=0.0):
        """ Process the request sent to the LLM
        """
        pass

    @abstractmethod
    def llm_generate(self,
                     prompt,
                     temperature=0.0):
        """  Generate a response based on the provided prompt
        """
        pass
```

A.3 Scheduler

For implementing schedulers, different scheduling strategies (e.g., FIFO, RR) will be achieved by creating specific schedulers that inherit from the base scheduler defined below. This approach ensures that new scheduling strategies can be added without interfering with existing schedulers, maintaining isolation and flexibility.

```python
class Scheduler:
    def __init__(
        self,
        llm,
        memory_manager,
        storage_manager,
        tool_manager,
        get_llm_request: LLMRequestQueueGetMessage,
        get_memory_request: MemoryRequestQueueGetMessage,
        get_storage_request: StorageRequestQueueGetMessage,
        get_tool_request: ToolRequestQueueGetMessage,
    ):
        """  Initializes the Scheduler with managers, request handlers, and threads for
            processing
        """
        self.get_llm_request = get_llm_request
        self.get_memory_request = get_memory_request
        self.get_storage_request = get_storage_request
        self.get_tool_request = get_tool_request
        self.active = False  # start/stop the scheduler
        self.log_mode = log_mode
        self.request_processors = {
            "llm_syscall_processor": Thread(target=self.run_llm_request),
            "mem_syscall_processor": Thread(target=self.run_memory_request),
            "sto_syscall_processor": Thread(target=self.run_storage_request),
            "tool_syscall_processor": Thread(target=self.run_tool_request)
        }
        self.llm = llm
        self.memory_manager = memory_manager
        self.storage_manager = storage_manager
        self.tool_manager = tool_manager

    def start(self):
        """  Starts the scheduler and runs all request processor threads
        """
        self.active = True
        for name, thread_value in self.request_processors.items():
            thread_value.start()

    def stop(self):
        """  Stops the scheduler and joins all processor threads
        """
        self.active = False
        for name, thread_value in self.request_processors.items():
            thread_value.join()

    def run_llm_syscall(self):
        """  Handles LLM system call requests
        """
        pass

    def run_memory_syscall(self):
        """  Handles memory system call requests
        """
        pass

    def run_storage_syscall(self):
        """  Handles storage system call requests
        """
        pass

    def run_tool_syscall(self):
        """  Handles tool system call requests
        """
        pass
```

A.4 Context Manager

The context manager is designed to interrupt the context during the generation process of LLM. In the context manager, we consider the following two cases. If the LLM successfully decodes the first token within the given time, the context manager will save the intermediate generation results during the decoding process. This ensures that intermediate results are stored without generating from the scratch. If the LLM is unable to decode the first token within the time slice, the context manager instead saves the intermediate KV caches generated during the pre-filling phase.

```
class ContextManager:
    def __init__(self):
        # Dictionary to store contexts by context ID
        self.context_data = {}
        # Lock to prevent concurrent access
        self.lock = threading.Lock()

    def gen_snapshot(self, cid, data):
        """  Acquire lock before modifying the dictionary
        """
        with self.lock:
            self.context_data[cid] = data

    def gen_restore(self, cid):
        """  Acquire lock before reading the dictionary
        """
        with self.lock:
            if self.check_restore(cid):
                return self.context_data[cid]
            else:
                return None

    def check_restore(self, cid):
        """  Acquire lock to ensure consistent read
        """
        with self.lock:
            return cid in self.context_data

    def clear_restore(self, cid):
        """  Acquire lock before modifying the dictionary
        """
        with self.lock:
            if process_id in self.context_data:
                del self.context_data[cid]
```

A.5 Memory Manager

In the memory manager, we leverage the zlip[1] library which is based on the prefix-tree data compression algorithm to convert the string-format data into compressed binary data and store. Implementation details are shown below.

[1] https://docs.python.org/3/library/zlib.html

```python
class MemoryManager:
    def __init__(self,
                 memory_limit,
                 eviction_k,
                 storage_manager):
        """  Initialize the memory manager with limits and a storage manager
        """
        self.memory_blocks = Dict()
        self.memory_limit = memory_limit
        self.eviction_k = eviction_k
        self.storage_manager = storage_manager

    def mem_alloc(self, aid):
        """  Allocate a new memory block for an agent if it does not exist
        """
        if aid not in self.memory_blocks:
            self.memory_blocks[aid] = OrderedDict()
            self.storage_manager.sto_create(aid)

    def mem_read(self, aid, rid):
        """  Retrieve a specific memory block for an agent, decompressing if in memory,
            otherwise from storage
        """
        if aid in self.memory_blocks and rid in self.memory_blocks[aid]:
            compressed_data = self.memory_blocks[aid].pop(rid)
            self.memory_blocks[aid][rid] = compressed_data
            return pickle.loads(zlib.decompress(compressed_data))
        else:
            return self.storage_manager.sto_read(aid, rid)

    def mem_write(self, aid, rid, s):
        """  Write a block of data for an agent, compressing and evicting if memory limit
            exceeded
        """
        self.mem_alloc(aid)
        serialized_data = pickle.dumps(s)
        compressed_data = zlib.compress(serialized_data)

        if rid in self.memory_blocks[aid]:
            self.memory_blocks[aid].pop(rid)
        self.memory_blocks[aid][rid] = compressed_data

        if self._total_memory_count() > self.memory_limit:
            self._evict_memory(aid)

    def mem_clear(self, aid):
        """  Clear all memory blocks for a specific agent, removing both from memory and
            storage
        """
        if aid in self.memory_blocks:
            del self.memory_blocks[aid]
            self.storage_manager.sto_clear(aid)

    def _total_memory_count(self):
        """  Calculate total memory block count across all agents
        """
        return sum(len(blocks) for blocks in self.memory_blocks.values())

    def _evict_memory(self, aid):
        """  Evict K least recently used blocks for an agent, storing them in persistent
            storage
        """
        if aid in self.memory_blocks:
            for _ in range(min(self.eviction_k,
                               len(self.memory_blocks[aid]))):
                rid, compressed_data = self.memory_blocks[aid].popitem(last=False)
                self.storage_manager.sto_write(aid, rid, pickle.loads(zlib.decompress(
                    compressed_data)))
```

A.6 Storage Manager

The storage manager uses files and vector databases (if vector database is enabled) to manage persistently stored data. Implementation details are shown below.

```python
class StorageManager:
    def __init__(self, storage_path, vector_db=None):
        """  Initializes storage path and optional vector database
        """
        self.storage_path = storage_path
        os.makedirs(self.storage_path, exist_ok=True)
        self.vector_db = vector_db

    def sto_create(self, aname, aid=None, rid=None):
        """  Creates a storage file and initializes a collection in the vector database
        """
        file_path = os.path.join(self.storage_path, f"{aid}_{rid}.dat" if aid and rid else
             f"{aname}.dat")

        if not os.path.exists(file_path):
            with open(file_path, "wb") as file:
                file.write(b"")
        if self.vector_db:
            self.vector_db.create_collection(f"{aid}_{rid}" if aid and rid else aname)

    def sto_read(self, aname, aid=None, rid=None):
        """ Reads data from a storage file if it exists
        """
        file_path = os.path.join(self.storage_path, f"{aid}_{rid}.dat" if aid and rid else
             f"{aname}.dat")

        if os.path.exists(file_path):
            with open(file_path, "rb") as file:
                compressed_data = file.read()
                return pickle.loads(zlib.decompress(compressed_data)) if compressed_data
                    else None
        return None

    def sto_write(self, aname, s, aid=None, rid=None):
        """  Writes compressed data to a storage file and adds it to the vector database
        """
        file_path = os.path.join(self.storage_path, f"{aid}_{rid}.dat" if aid and rid else
             f"{aname}.dat")

        with open(file_path, "ab") as file:
            compressed_data = zlib.compress(pickle.dumps(s))
            file.write(compressed_data)
        if self.vector_db:
            self.vector_db.add(f"{aid}_{rid}" if aid and rid else aname, s)

    def sto_clear(self, aname, aid=None, rid=None):
        """  Clears the storage file and deletes the corresponding vector database
            collection
        """
        file_path = os.path.join(self.storage_path, f"{aid}_{rid}.dat" if aid and rid else
             f"{aname}.dat")

        if os.path.exists(file_path):
            os.remove(file_path)
        if self.vector_db:
            self.vector_db.delete(f"{aid}_{rid}" if aid and rid else aname)

    def sto_retrieve(self, aname, query, aid=None, rid=None):
        """  Retrieves data from the vector database using a query
        """
        if self.vector_db:
            return self.vector_db.retrieve(f"{aid}_{rid}" if aid and rid else aname, query
                )
        return None
```

A.7 Tool Manager

The tool manager module is responsible for loading tools executing tools with tool conflict prevention mechanisms. Implementation details are shown below.

```
class ToolManager:
    def __init__(self):
        """  Initializes the ToolManager with a conflict map and a thread lock
        """
        self.tool_conflict_map = {}
        self.lock = threading.Lock()

    def tool_run(self, syscall) -> None:
        """  Runs a tool with given parameters, ensuring no conflict with other tool
            executions
        """
        request_data = syscall.request_data
        tool_org_and_name, tool_params = request_data["name"], request_data["paramemters"]

        if tool_org_and_name not in self.tool_conflict_map.keys():
            with self.lock:
                self.tool_conflict_map[tool_org_and_name] = 1
                tool_class = self.load_tool_instance(tool_org_and_name)

                tool = tool_class(
                    tool_org_and_name=tool_org_and_name.split("/")[1]
                )
                tool.run(
                    params=tool_params
                )

            with self.lock:
                self.tool_conflict_map.pop(tool_org_and_name)

    def snake_to_camel(self, snake_str):
        """  Converts a snake_case string to CamelCase
        """
        components = snake_str.split("_")
        return "".join(x.title() for x in components)

    def load_tool_instance(self, tool_org_and_name):
        """  Dynamically loads a tool instance based on organization and tool name
        """
        org, tool_name = tool_org_and_name.split("/")
        module_name = ".".join(["tools", org, tool_name])
        class_name = self.snake_to_camel(tool_name)

        tool_module = importlib.import_module(module_name)
        tool_instance = getattr(tool_module, class_name)
        return tool_instance
```

A.8 Access Manager

The access manager provides two key functions: First is to check access when agents attempt to access other agents' resources. Second is to request user permission before agents execute irreversible actions such as deletion of files. Implementation details are shown below.

```python
class AccessManager:
    def __init__(self):
        self.privilege_map = {}

    def add_privilege(self, sid, tid):
        """ Assigns an agent into another agent's privilege group
        """
        if tid not in self.privilege_map.keys():
            self.privilege_map[tid] = []
        self.privilege_map[tid].append(sid)

    def check_access(self, sid, tid):
        """  Checks if the source agent is in the target agent's priviledge group
        """
        if sid in self.privilege_map.get(sid):
            return True
        else:
            return False

    def ask_permission(self, aid, operation):
        """  Prompts the user for confirmation before an irreversible operation
        """
        confirmation = input(f"Confirm?␣(yes/no):").strip().lower()
        return confirmation == "yes"
```

A.9 Module Hooks

To effectively separate the interface of calling the AIOS kernel modules from the implementation details, we employ a hook mechanism to initialize modules and export the necessary call interfaces. Here are the hooks we use for initializing modules.

```python
@validate(LLMParams)
def useLLM(params: LLMParams) -> LLM:
    """  Initialize and return a Language Learning Model (LLM) instance.

    Args:
        params (LLMParams): Parameters required for LLM initialization.

    Returns:
        LLM: An instance of the initialized LLM.
    """
    return LLM(**params.model_dump())
```

```python
@validate(MemoryManagerParams)
def useMemoryManager(params: MemoryManagerParams) -> MemoryManager:
    """  Initialize and return a memory instance.

    Args:
        params (MemoryParams): Parameters required for Memory Manager Initialization.

    Returns:
        Memory Manager: An instance of the initialized Memory Manager.
    """
    return MemoryManager(**params.model_dump())
```

```python
@validate(StorageManagerParams)
def useStorageManager(params: StorageManagerParams) -> StorageManager:
    """  Initialize and return a storage instance.

    Args:
        params (StorageManagerParams): Parameters required for Memory Manager
            Initialization.

    Returns:
        Storage Manager: An instance of the initialized Storage Manager.
    """
    return StorageManager(**params.model_dump())
```

```python
@validate(ToolManagerParams)
def useToolManager(params: ToolManagerParams) -> ToolManager:
    """  Initialize and return a tool instance.

    Args:
        params (ToolManagerParams): Parameters required for Tool Manager Initialization.

    Returns:
        Tool Manager: An instance of the initialized Tool Manager.
    """
    return ToolManager(**params.model_dump())
```

```
@validate(SchedulerParams)
def useScheduler(params: SchedulerParams,) -> Tuple[Callable[[], None], Callable[[], None
    ]]:
    """  Initialize and return a scheduler with start and stop functions.

    Args:
        params (SchedulerParams): Parameters required for the scheduler.

    Returns:
        Tuple: A tuple containing the start and stop functions for the scheduler.
    """
    if params.get_llm_request is None:
        from aios.hooks.stores._global import global_llm_req_queue_get_message
        params.get_llm_request = global_llm_req_queue_get_message

    if params.get_memory_request is None:
        from aios.hooks.stores._global import global_memory_req_queue_get_message
        params.get_memory_request = global_memory_req_queue_get_message

    if params.get_storage_request is None:
        from aios.hooks.stores._global import global_storage_req_queue_get_message
        params.get_storage_request = global_storage_req_queue_get_message

    if params.get_storage is None:
        from aios.hooks.stores._global import global_tool_req_queue_get_message
        params.get_tool_request = global_tool_req_queue_get_message

    scheduler = Scheduler(**params.model_dump())

    # Function to start the scheduler
    def startScheduler():
        scheduler.start()

    # Function to stop the scheduler
    def stopScheduler():
        scheduler.stop()

    return startScheduler, stopScheduler
```

B AIOS-Agent SDK

B.1 Query and Response

In the AIOS-Agent SDK, two main data structures, Query and Response are defined to facilitate agent interactions with the AIOS kernel by structuring input requests and output responses. The Query class serves as the input structure for agents to perform various actions within AIOS. It includes: The Response class represents the output structure that agents receive after the AIOS kernel processes the functions to return the Response.

```python
class Query(BaseModel):
    """  Query class represents the input structure for performing various actions.

    Attributes:
        messages: A list of dictionaries where each dictionary
            represents a message containing 'role' and 'content' or other key-value pairs.
        tools: An optional list of JSON-like objects (dictionaries)
            representing tools and their parameters. Default is an empty list.
        action_type: A string that must be one of "chat", "call_tool", or "operate_file".
            This restricts the type of action the query performs.
        message_return_type: The type of the response message. Default is "text".
    """

    messages: List[Dict[str, Union[str, Any]]]
    # List of message dictionaries, each containing role and content.

    tools: Optional[List[Dict[str, Any]]] = Field(default_factory=list)
    # List of JSON-like objects (dictionaries) representing tools.

    action_type: Literal["chat", "call_tool", "operate_file"] = Field(default="chat")
    # Restrict the action_type to specific choices.

    message_return_type: str = Field(default="text")
    # Type of the return message, default is "text".
```

```python
class Response(BaseModel):
    """  Response class represents the output structure after performing actions.

    Attributes:
        response_message (Optional[str]): The generated response message. Default is None.
        tool_calls (Optional[List[Dict[str, Any]]]): An optional list of JSON-like objects
             (dictionaries)
            representing the tool calls made during processing. Default is None.
    """
    response_message: Optional[str] = None
    # The generated response message, default is None.

    tool_calls: Optional[List[Dict[str, Any]]] = None
    # List of JSON-like objects representing tool calls, default is None.
```

B.2 Native Supported External Tools.

As is illustrated in Table 6, AIOS-Agent SDK integrates diverse computational tools to address a wide spectrum of information processing tasks. The SDK incorporates 17 native tools spanning multiple modalities and functionalities, enabling sophisticated interaction patterns across text, image, and audio domains. These tools can be categorized into three primary sources: established technology providers (Google, Bing, WolframAlpha), specialized API hubs (Rapid API Hub), and advanced AI model providers (Huggingface). The toolkit's architecture demonstrates particular strength in text-based operations, with 12 tools supporting text input or output modalities. This includes fundamental information retrieval services (Arxiv, BingSearch, Wikipedia), specialized analytical tools (CurrencyConverter, MoonPhaseSearch), and domain-specific applications (ImdbRank, TripAdvisor). Furthermore, the SDK exhibits robust cross-modal capabilities through tools like VisualQuestionAnswering (image-text integration), TextToAudio (text-to-speech synthesis), and VoiceActivityRecognition (speech-to-text conversion).

B.3 Native Agent Examples

Here we provide examples of agents developed by leveraging the capabilities of the AIOS-Agent SDK. **Travel Agent:** The travel agent utilizes the AIOS-Agent SDK to access APIs and tools for trip planning, including searching for flights, accommodations, and local activities. **Rec Agent:** The recommendation agent leverages the AIOS-Agent SDK to suggest movies and TV series. **Math Agent:** This agent utilizes the SDK to access mathematical tools and computation resources, allowing it to solve equations, perform calculations, and provide step-by-step explanations for different math problems.

Table 6. Native tools supported by AIOS-Agent SDK, ordered by names in alphabet.

Tool Name	Source	Type	Modality (Input → Output)
Arxiv	Arxiv	API	Text → Text
BingSearch	Bing	API	Text → Text
CurrencyConverter	Rapid API Hub	API	Text → Text
GooglePlace	Google	API	Image/Text → Text
GoogleSearch	Google	API	Text → Image
ImageCaption	Huggingface	Local Model/API	Text → Text
ImdbRank	Rapid API Hub	API	Text → Text
MoonPhaseSearch	Rapid API Hub	API	Text → Text
Shazam	Rapid API Hub	API	Text → Text/Audio
TextToAudio	Huggingface	Local Model/API	Text → Audio
TextToImage	Huggingface	Local Model/API	Text → Image
TripAdvisor	Rapid API Hub	API	Text → Text
VisualQuestionAnswering	Huggingface	Local Model/API	Image & Text → Text
VoiceActivityRecognition	Huggingface	Local Model/API	Audio → Text
Wikipedia	Wikipedia	API	Text → Text
WolframAlpha	WolframAlpha	API	Text → Text
WordsAPI	Rapid API Hub	API	Text → Text

Creation Agent: The creation agent is tailored for content generation tasks, such as writing, graphic design, or even video editing. By accessing creative tools and resources through the AIOS-Agent SDK, the creation agent can assist with generating textual content, designing visuals, or assembling multimedia elements, enabling users to produce high-quality content efficiently. **Academic Agent:** The academic agent is designed to support research and learning, utilizing the SDK to access scholarly articles to assist with literature reviews and even provide explanations on complex academic topics.

TravelAgent Profile

Description: You are an expert in planning and managing travel itineraries.

Workflow:

1. Identify the destination and search for hotel locations using the hotel_location_search tool.
2. Based on the hotel locations, find suitable hotels using the hotel_search tool, and select the best one.
3. Get detailed information about the selected hotel using the get_hotel_details tool.
4. Search for the nearest airport to the origin using the airport_search tool.
5. Search for the nearest airport to the destination using the airport_search tool.
6. Find available flights to the destination airport using the flight_search tool using the correct date.
7. Search for restaurant locations near destination using the restaurant_location_search tool.
8. Based on the restaurant locations, find suitable restaurants using the restaurant_search tool.
9. Get detailed information about the selected restaurants using the get_restaurant_details tool.
10. Gather additional relevant information about the destination the user is visiting using the wikipedia tool.
11. Integrate the information gathered from the previous steps to provide a comprehensive travel plan.

Available tools:

1. TripAdvisor
2. Wikipedia

Example of task inputs: I want to take a trip to Paris, France from July 4th to July 10th, 2024, and I am traveling from New York City. Help me plan this trip.

RecAgent Profile

Description: You are an expert who is good at recommending TV series and movies.
Workflow:
1. Identify the tool that you need to call to obtain information.
2. Based on the information, give recommendations for the user based on the constrains.

Available tools:
1. TripAdvisor
2. Wikipedia

Example of task inputs: Recommend three action movies from the past five years ranked between 1 and 20 with ratings above 8.0.

CreationAgent Profile

Description: You are an expert who is good at content creation.
Workflow:
1. Convert the vague description of the content requirements into concrete objects and fill in more details.
2. Identify the tool to call the tool to create content based on the filled details.

Available tools:
1. SDXL-Turbo

Example of task inputs: Create an image of a sleek, high-tech futuristic city with a vibrant nightlife atmosphere.

MathAgent Profile

Description: You are an expert who is good at solving mathematical problems.

Workflow:
1. Identify the tool to call to do some pre-calculation.
2. Perform mathematical operations using the pre-calculated result, which could involve addition, subtraction, multiplication, or division with other numeric values to solve the problem.

Available tools:
1. Currency Converter
2. WolframAlpha

Example of task inputs: Convert 15000 MXN to Canadian Dollars and find out how much it would be in USD if 1 CAD equals 0.79 USD.

AcademicAgent Profile

Description: You are an expert who is good at looking up and obtaining information from academic articles.

Workflow:
1. Identify the tool to call based on the academic requirements and call the tool.
2. Gather the information obtained from the tool to write an outline or summarization.

Available tools:
1. Arxiv API

Example of task inputs: Summarize recent studies on the role of artificial intelligence in drug discovery from 2018 to 2023.

B.4 Support of Agent Frameworks

The core idea to adapt agents built by existing agent frameworks for AIOS is to identify the core functions that will interact with system resources and change that with functions in our native adapters. In this section, we illustrate the crucial adaptation function that needs to be changed to run agents built by other agent frameworks on AIOS.

ReAct (Yao et al., 2023). The ReAct framework integrates reasoning and action steps in language models, allowing them

to generate intermediate reasoning traces alongside actionable steps for complex task completion. This dual approach helps models not only plan and track their thought process but also interact with external tools, improving performance on tasks like question answering, game environments, and decision-making problems that require multi-step reasoning and adaptability. By alternating between reasoning and action, ReAct reduces errors from solely predictive responses and enables more accurate, contextually aware task completion.

Reflexion (Shinn et al., 2023). The Reflexion framework enhances language agents with a feedback-driven mechanism, allowing them to learn from mistakes and adapt behavior through self-reflective feedback loops. By leveraging verbal reinforcement learning, agents assess and adjust their actions, which improves performance on complex tasks through iterative learning. This approach makes language agents more resilient and adaptive, enabling them to handle tasks with evolving requirements and uncertainty.

Autogen (Wu et al., 2023). AutoGen introduces a framework that leverages multiple language model agents with distinct roles (such as Planner, Executor, and Reflector) to collaboratively solve complex tasks through structured, goal-oriented conversations. By enabling agents to communicate and share intermediate results, AutoGen coordinates multi-step processes like data analysis, decision-making, and iterative problem-solving, significantly enhancing efficiency and accuracy beyond a single model's capabilities. This approach empowers next-generation applications, allowing LLMs to tackle dynamic workflows, adapt to task-specific nuances, and achieve higher performance in real-world scenarios. Below is the code of adapting Autogen for AIOS. Due to ongoing refactoring work by the Autogen team, only Autogen-0.2 (the latest stable version) is supported.

```
@add_framework_adapter("AutoGen~0.2")
def prepare_autogen_0_2():
    """
    Replace OpenAIWrapper and ConversableAgent methods with aios's implementation.

    This function is used to adapt autogen's API to aios's API, and it is used
    internally by aios.
    """
    # Replace OpenAIWrapper method
    OpenAIWrapper.__init__ = adapter_autogen_client_init
    OpenAIWrapper.create = adapter_client_create
    OpenAIWrapper.extract_text_or_completion_object =
        adapter_client_extract_text_or_completion_object

    # Replace agent method
    ConversableAgent._print_received_message = _adapter_print_received_message
    ConversableAgent._generate_oai_reply_from_client =
        _adapter_generate_oai_reply_from_client
    ConversableAgent.generate_tool_calls_reply = adapter_generate_tool_calls_reply
    ConversableAgent.execute_function = adapter_execute_function
    ConversableAgent._a_execute_tool_call = _adapter_a_execute_tool_call
    ConversableAgent.update_tool_signature = adapter_update_tool_signature
    ConversableAgent.__init__ = adapter_autogen_agent_init
```

Open-Interpreter (Lucas, 2024). Open Interpreter is an open-source framework that enables users to interact with LLMs through a ChatGPT-like interface to interpret and execute complex instructions across programming languages directly in the terminal. It supports both locally-hosted and cloud-based LLMs, allowing for streamlined code execution and debugging in natural language. By translating natural language instructions into executable code, Open Interpreter offers an intuitive environment that not only simplifies development workflows but also facilitates learning by providing detailed explanations and interactive support for various coding challenges, making it suitable for developers at all skill levels. Below is the core function to be adapted for Open-Interpreter.

```
@add_framework_adapter("Open-Interpreter")
def prepare_interpreter():
    """  Prepare the interpreter for running LLM in aios.
    """

    # Set the completion function in the interpreter
    interpreter.llm.completions = adapter_aios_completions
```

MetaGPT (Hong et al., 2023). MetaGPT proposes a meta-programming approach that optimizes LLM-driven multi-agent systems by integrating task-oriented programming paradigms for complex, collaborative problem-solving. MetaGPT encodes Standardized Operating Procedures (SOPs) directly into structured prompt sequences, creating streamlined workflows that empower agents with human-like domain expertise to systematically verify intermediate outputs and proactively mitigate errors. Along this line, MetaGPT addresses the limitations of existing LLM-based frameworks, such as hallucination and cascading errors during agent chaining. This framework facilitates the decomposition of complex tasks into manageable, interdependent subtasks, improving overall system robustness, especially in high-stakes, iterative processes where reliability across agent interactions is crucial. Below is the core function to be adapted for MetaGPT.

```
@add_framework_adapter("MetaGPT")
def prepare_metagpt():
    """
    Prepare the metagpt module to run on aios.

    This function does the following:
    1. Create a fake configuration file with effects similar to "metagpt --init-config"
    2. Replace the llm used in metagpt with aios_call
    """
    # create fake configuration file
    prepare_metagpt_config()

    BaseLLM.aask = adapter_aask
```

C Details of Agent Benchmarks

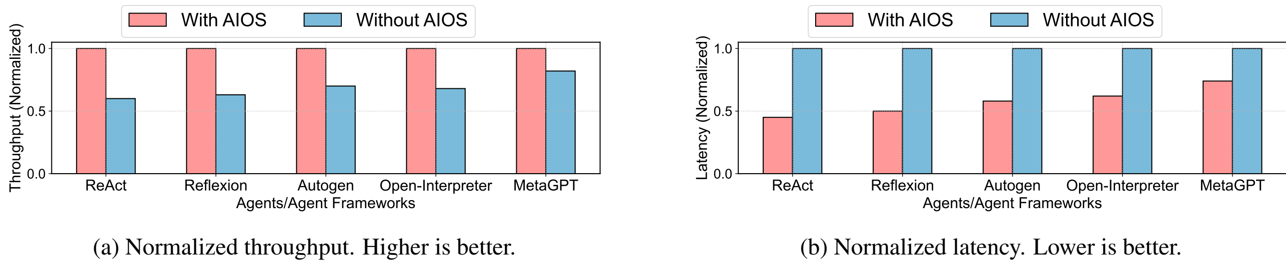
C.1 HumanEval

The authors (Chen et al., 2021b) [2] introduced HumanEval, a benchmark dataset comprising 164 handwritten programming problems for evaluating functional correctness of code generation models. Each problem consists of a function signature, docstring, implementation body, and comprehensive test suite, with an average of 7.7 test cases per problem. The handwritten nature of these problems is crucial, given that modern language models are typically trained on large portions of GitHub code containing existing solutions to programming challenges and contest problems. HumanEval is designed to assess multiple aspects of code generation capability: natural language comprehension, logical reasoning, algorithmic thinking, and mathematical operations. Through this publicly available benchmark, researchers can conduct rigorous and standardized evaluations of code generation models.

C.2 MINT

MINT (Wang et al., 2023b)[3] introduced a benchmark to evaluate LLMs' ability to solve challenge tasks through multi-turn interactions. The benchmark focuses on code generation, decision making, and reasoning tasks that require LLMs to utilize tools and incorporate natural language feedback. MINT was constructed by curating multiple single-turn datasets, reducing an original collection of 29,307 instances to 586 carefully selected examples. The benchmark uses success rate (SR) as its primary evaluation metric, measuring the percentage of successfully completed tasks. For a given interaction limit k ranging from 1 to 5, each LLM is allowed up to k turns of interaction, with performance measured as SR_k. In our experiments, we set $k = 5$ and focus exclusively on MINT's code generation subset.

[2] The dataset can be found at https://www.github.com/openai/human-eval.
[3] https://xwang.dev/mint-bench/



(a) Normalized throughput. Higher is better.

(b) Normalized latency. Lower is better.

Figure 10. Efficiency analysis on different agent frameworks evaluated on the Llama-3.1-8b model on the MINT benchmark.

C.3 GAIA

General AI Assistant (GAIA) (Mialon et al., 2023)[4] is a benchmark designed to represent a significant milestone in AI research by evaluating fundamental capabilities essential for general intelligence. Unlike traditional benchmarks that focus on specialized professional knowledge, GAIA emphasizes everyday tasks that require core abilities including logical reasoning, multi-modal processing, web navigation, and effective tool utilization. GAIA comprises 466 questions that evaluate AI assistants across multiple capabilities including reasoning, multi-modal understanding, coding, and tool usage (particularly web browsing), with tasks involving various data formats like PDFs, spreadsheets, images, videos, and audio. The benchmark organizes questions into three difficulty levels based on the number of required steps and tools: Level 1 requires minimal tool usage ($\leq$ 5 steps), Level 2 demands multiple tools and 5-10 steps, while Level 3 tests advanced general assistance capabilities through complex, multi-step sequences requiring diverse tool combinations. Additionally, while web browsing is central to GAIA, the benchmark deliberately excludes complex web interactions like file uploads or posting comments, leaving such evaluations for future research.

C.4 SWEBench-Lite

SWE-bench (Jimenez et al., 2024)[5] is a software engineering benchmark constructed through a rigorous three-stage pipeline that processes GitHub pull requests (PRs) from 12 popular Python repositories. The pipeline filters approximately 90,000 PRs based on attributes (issue resolution and test contribution) and execution criteria (successful installation and fail-to-pass test transitions), resulting in 2,294 high-quality task instances. Each task requires models to generate patch files that resolve software issues, with success determined by comprehensive test coverage. The benchmark distinguishes itself through real-world challenges, extensive input context (averaging 195 words per issue), cross-context editing requirements (typically spanning 1.7 files and 32.8 lines per solution), and robust test-based evaluation. Notably, SWE-bench's automated collection process enables continuous updates with new task instances from GitHub repositories, ensuring benchmark relevance over time.

D Additional Experimental Results

D.1 Efficiency analysis.

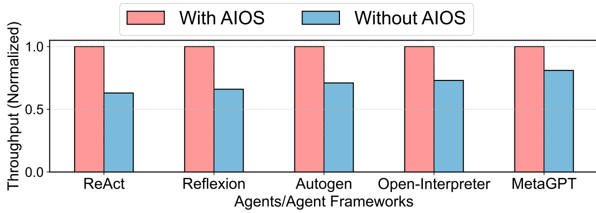
We also report the throughput and latency of running agents on other three benchmarks on Llama-3.1-8b and Mistral-7b compared between using AIOS and without using AIOS. The results are shown in Figure 10 and Figure 11, Figure 12 and Figure 13, Figure 14 and Figure 15, respectively.

D.2 Correctness of context switch.

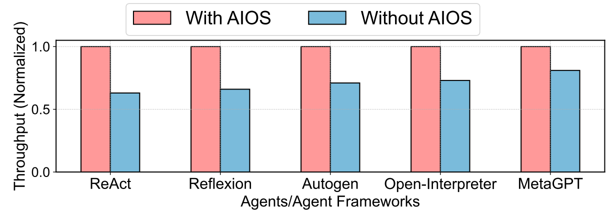
To assess the correctness of the context switch supported by the context manager, we employ the BLEU score (Papineni et al., 2002) and BERT score (Zhang et al., 2019) to measure text similarity. The similarity is calculated against the final outputs generated for the same agent under the same conditions, only varying with context switch enabled and disabled. As demonstrated in Table 7, both BLEU and BERT scores achieve a value of 1.0. The suggests that the context switch does not introduce discrepancies in output quality, suggesting the reliability of the AIOS.

[4] https://huggingface.co/gaia-benchmark

[5] https://www.swebench.com/

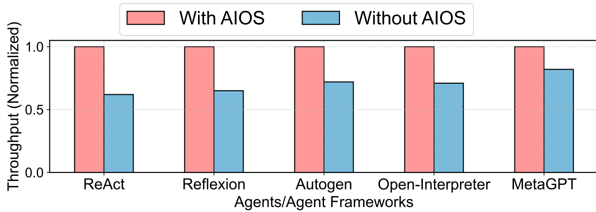


(a) Normalized throughput. Higher is better.

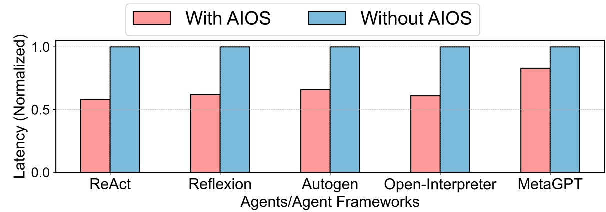


(b) Normalized latency. Lower is better.

Figure 11. Efficiency analysis on different agent frameworks evaluated on the Mistral-7b model on the MINT benchmark.



(a) Normalized throughput. Higher is better.



(b) Normalized latency. Lower is better.

Figure 12. Efficiency analysis on different agent frameworks evaluated on the Llama-3.1-8b model on the GAIA benchmark.

E DISCUSSION

E.1 Ethical Consideration

In this section, we discuss both potential positive and negative societal impacts of the work.

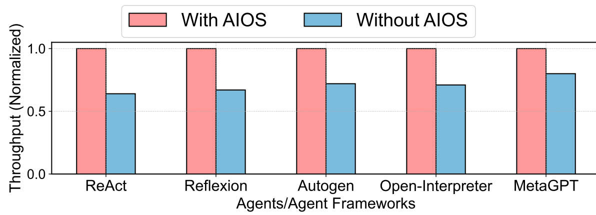
The potential positive societal impacts include: 1) Enhanced efficiency and productivity: AIOS can automate routine tasks, achieve more efficient operations, optimize resource allocation, and reduce bottlenecks, leading to better service and improved efficiency for agent developers; 2) Improved user experience: with better context, memory, and storage management, AIOS can offer more personalized and responsive interactions, enhancing user satisfaction across various applications; 3) Innovation ecosystem: the creation of AIOS could foster a vibrant ecosystem of agent developers and researchers, driving innovation in AI technologies and applications.

The potential negative societal impacts include: 1) Privacy concerns: the integration of LLMs into operating systems may raise privacy concerns, as AI models such as LLMs may require access to personal data to provide effective services; 2) Security risks: as AI systems become more integral to critical infrastructure, they could become targets for cyberattacks, potentially compromising sensitive data and operations; 3) System failures: the failure of integrated systems could have widespread consequences, affecting multiple sectors simultaneously and causing disruptions.

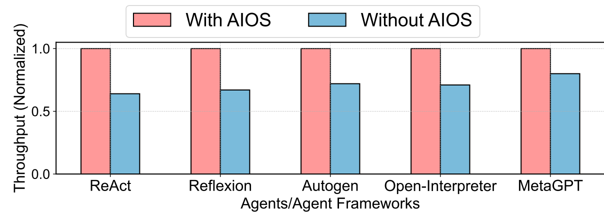
Balancing the impacts: To maximize the positive impacts and mitigate the negative ones, it is crucial to adopt a balanced approach to the development and deployment of AIOS, such as 1) Rules and standards: Implementing responsible development rules and standards to ensure data privacy, security, and ethical use of AI; 2) Robust design: implementing

Table 7. Correctness of context switch (text-based and logits-based), which checks the similarity between the generated final outputs with context switch enabled and disabled.

LLM core	Method	BLEU Score	BERT Score
Mistral-7B	Text-based	1.0	1.0
	Logits-based	1.0	1.0
Llama-3-8B	Text-based	1.0	1.0
	Logits-based	1.0	1.0

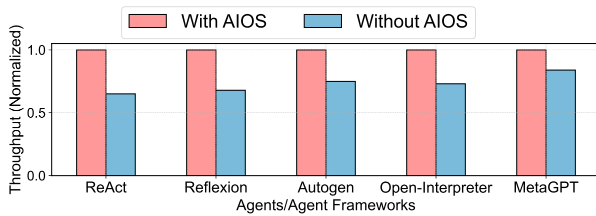


(a) Normalized throughput. Higher is better.

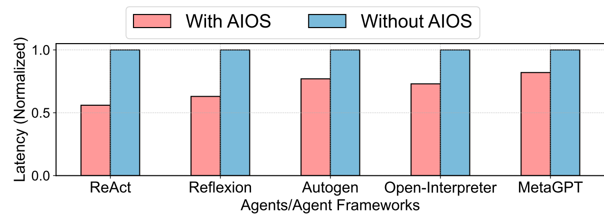


(b) Normalized latency. Lower is better.

Figure 13. Efficiency analysis on different agent frameworks evaluated on the Mistral-7b model on the GAIA benchmark.



(a) Normalized throughput. Higher is better.



(b) Normalized latency. Lower is better.

Figure 14. Efficiency analysis on different agent frameworks evaluated on the Llama-3.1-8b model on the SWE-Bench-Lite benchmark.

robust system design, regular maintenance, comprehensive testing, continuous monitoring, backup and recovery plans, developer training, careful documentation, clear communication, and leveraging AI for predictive maintenance and automated recovery; 3) Public engagement: engaging with the public to raise awareness about the benefits and challenges of AI, ensuring that societal concerns are addressed in the development process.

By addressing these considerations, society can harness the potential of AIOS while mitigating its risks, leading to a more equitable and prosperous future.

E.2 Future Directions

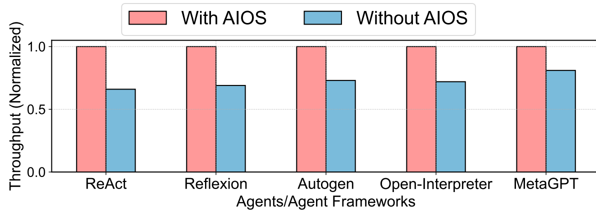
With AIOS as a foundation, there are many directions for future research to pursue. This section outlines potential areas of study that expand upon the foundational features of AIOS.

Semantic Scheduling Algorithms. The scheduling function of AIOS lays the groundwork for the development of more advanced algorithms. Future research could focus on algorithms that perform dependency analysis among agent requests, optimizing the allocation of computational resources. Additionally, some of the tool resources are locally deployed models, which can also be incorporated into the scheduling paradigm. This includes the management of tool status and snapshots, suggesting a move towards a unified scheduling framework that encompasses both agents and their tools.

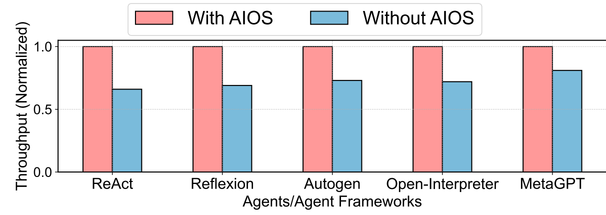
Efficiency of Context Management. More efficient mechanisms can be devised to assist context management. For example, the pursuit of time-efficient context management techniques could significantly augment user experience by expediting the processes of context snapshotting and restoration. Also, context compression techniques can also be leveraged prior to snapshotting, which can yield a more space-efficient solution.

Optimization of Memory and Storage Architecture. In the context of agent collaboration and communication, the future design of memory and storage systems can adopt a shared approach, enabling the sharing of memory and storage between agents. Such an architecture would enable agents to access a communal pool of memory and storage, thereby improving the agents' decision-making ability since one agent can benefit from other agents' memory or storage. Moreover, future work can explore hierarchical storage solutions, designed to optimize data retrieval and storage efficiency. This could involve prioritizing quicker access and reduced storage allocation for frequently accessed data, and vice versa for less frequently accessed information.

Safety and Privacy Enhancements. The aspect of safety in AIOS necessitates protective measures against various attacks,



(a) Normalized throughput. Higher is better.



(b) Normalized latency. Lower is better.

Figure 15. Efficiency analysis on different agent frameworks evaluated on the Mistral-7b model on the SWE-Bench-Lite benchmark.

ensuring the system's resilience against malicious attacks, such as jailbreaking of LLM or unauthorized access of other agents' memory. In the realm of privacy, the exploration of advanced encryption techniques is vital for safeguarding data transmission within AIOS, thus maintaining the confidentiality of agent communications. Furthermore, the implementation of watermarking techniques could serve to protect the intellectual property of agent developers by embedding unique identifiers in outputs, facilitating the tracing of data lineage.

In a nutshell, AIOS stands as a motivating body of work that brings a broad spectrum of research opportunities. Each outlined direction not only can build upon the foundational elements of AIOS but also can contribute to the advancement of the field at large.