AIOS: LLM Agent Operating System

Kai Mei
Rutgers University

Zelong Li
Rutgers University

Shuyuan Xu
Rutgers University

Ruosong Ye
Rutgers University

Yingqiang Ge
Rutgers University

Yongfeng Zhang
Rutgers University

Abstract

The integration and deployment of large language model (LLM)-based intelligent agents have been fraught with challenges that compromise their efficiency and efficacy. Among these issues are sub-optimal scheduling and resource allocation of agent requests over the LLM, the difficulties in maintaining context during interactions between agent and LLM, and the complexities inherent in integrating heterogeneous agents with different capabilities and specializations. The rapid increase of agent quantity and complexity further exacerbates these issues, often leading to bottlenecks and sub-optimal utilization of resources. Inspired by these challenges, this paper presents AIOS, an LLM agent operating system, which embeds large language model into operating systems (OS) as the brain of the OS, enabling an operating system "with soul"—an important step towards AGI. Specifically, AIOS is designed to optimize resource allocation, facilitate context switch across agents, enable concurrent execution of agents, provide tool service for agents, and maintain access control for agents. We present the architecture of such an operating system, outline the core challenges it aims to resolve, and provide the basic design and implementation of the AIOS. Our experiments on concurrent execution of multiple agents demonstrate the reliability and efficiency of our AIOS modules. Through this, we aim to not only improve the performance and efficiency of LLM agents but also to pioneer for better development and deployment of the AIOS ecosystem in the future. The project is open-source at `https://github.com/agiresearch/AIOS`.

1 Introduction

In the field of autonomous agents, research endeavors [1, 2, 3] are directed towards systems that can operate independently, make decisions, and perform tasks with no or minimal human intervention. These agents are designed to understand instructions, process information, make decisions and take action to achieve a state of autonomy. The advent of large language models (LLMs) [4, 5, 6] has brought new possibilities to the agent development [7]. Current LLMs have shown great power in understanding instructions [8, 9, 10, 11], reasoning and solving problems [12, 13, 14, 15, 16], and interacting with human users [17] as well as external environments [18, 19]. Built upon these powerful LLMs, emergent LLM-based agents [7, 20, 21, 22] 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 compelling example of how an LLM-based 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 sequentially to book flights, reserve hotels, process payments, and update calendars based on the user's preferences. During the plan execution, agents

*Author Affiliations: Department of Computer Science, Rutgers University, New Brunswick, NJ 08854; **Author Emails**: kai.mei, zelong.li, shuyuan.xu, ruosong.ye, yingqiang.ge, yongfeng.zhang@rutgers.edu

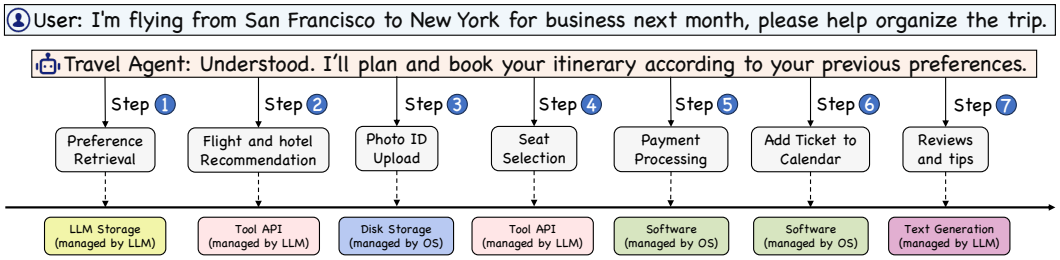

Figure 1: A motivating example of how an agent (i.e., Travel Agent) requires both LLM level and OS level resources and functions to complete a task.

show the reasoning and decision-making abilities, 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 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).

Accompanied by the exponential growth in the agent quantity and complexity, there is an increasing strain on the functionalities of LLM and OS. For example, scheduling and prioritizing agent requests in limited LLM resources poses a significant challenge. Moreover, the LLM's generation process can become time-intensive when dealing with lengthy contexts, occasionally resulting in the generation being suspended by the scheduler. This raises the problem of devising a mechanism to snapshot the LLM's current generation result, thereby enabling pause/resume behavior even when the LLM has not finalized the response generation for the current request. Furthermore, once an agent has obtained the list of available calling tools, determining the optimal sequence for invoking these tools presents yet another challenge since multiple agents may need to call the same tool. Additionally, the concurrent operation of multiple agents necessitates a robust system for memory management across different agents, while also ensuring stringent enforcement of privacy and access control measures.

To address the above mentioned challenges, we propose AIOS, an LLM agent operating system (Figure 2) to provide module isolation and aggregations of LLM and OS functionalities. To address the potential conflicts arising between tasks associated with LLM and those unrelated to LLM, we propose the design of an LLM-specific kernel. This kernel segregates the OS-like duties, particularly those related to the oversight of LLM agents, their corresponding resources, and development toolkits. Through this segregation, the LLM kernel aims to enhance the management and coordination of LLM-related activities. Within the proposed LLM kernel, we have devised a suite of modules, each dedicated to addressing the distinct functions pertinent to LLM operations. An overview of these modules and their respective functionalities is presented in the following, with the comprehensive discussion of their underlying mechanisms detailed in Section 4.

- **Agent Scheduler**: Prioritizes and schedules agent requests to optimize LLM utilization.
- **Context Manager**: Supports snapshot and restore the intermediate generation status in LLM and context window management of LLM.
- **Memory Manager**: Provides short-term memory for each agent's interaction logs.
- **Storage Manager**: Persists agent interaction logs to long-term storage for future retrieval.
- **Tool Manager**: Manages agent's calling of external API tools (e.g., search, scientific computing).
- **Access Manager**: Enforces privacy and access control policies between agents.

Aside from the modules, the kernel exposes an LLM system call interface through which agents can transparently leverage these services. Moreover, we design the AIOS SDK to further encapsulate the LLM system calls, providing more convenient agent library functions for agent developers. With the AIOS architecture, an agent like the travel planner can break down its task into steps that fluidly combine LLM reasoning (e.g., plan generation and tool calling decision) and OS-level actions (e.g., accessing storage and executing software services). This synergistic combination of capabilities equips multiple LLM agents to tackle increasingly complex, multi-modal tasks that require reasoning, execution, and interaction with the physical world.

Looking ahead, we envision extending the AIOS to support even tighter agent-world integration (e.g., through robotic control), more intelligent resource management, and safer multi-agent collaboration. Ultimately, AIOS serves as the crucial platform to facilitate the development, deployment and usage of various complex LLM agents.

2 Related Work

2.1 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. Initially, operating systems served to bridge the gap between the user-level tasks and the binary functionality of computer hardware, such as electron and gate manipulation. Their evolution saw a transition from simple batch job processing [23] to more advanced process management techniques like time-sharing [24] and multi-task processing [25, 26], which facilitated the handling of increasingly complex tasks. The progress moved toward modularization within the OS, delineating specific responsibilities such as process scheduling [27, 28], memory management [29, 30], and filesystem management [31, 32], enhancing efficiency and manageability. The further advent of graphical user interfaces (GUIs), e.g., Macintosh[1], Windows[2] and GNOME[3], 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 [33]. Notable examples of OS ecosystems include Android Studio[4], XCode[5] and Cloud SDK[6]. 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. Nowadays, we stand at the transformative phase to see the potential of intelligent operating systems. 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.

2.2 Large Language Model Agents

Large language model (LLM) based autonomous agents take natural language instructions as input for complex task solving. The research on LLM-based agents can be generally classified into single-agent systems and multi-agent systems [33].

LLM-based Single-Agent Systems. LLM-based single-agent systems (SAS) use a single LLM agent for complex task solving, such as travel planning, personalized recommendation, and artistic design [7]. 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 [7, 34, 35, 36, 37], browse websites [38, 22], or execute codes [39], while agents in the physical environment may manipulate objects [19, 40, 41], carry out lab experiments [42, 43], or make actionable decisions [44, 45].

LLM-based Multi-Agent Systems. LLM-based multi-agent systems (MAS) leverage the interaction among multiple agents for problem solving. The relationship among the multiple agents could be cooperative, competitive, or a mixture of cooperation and competition [33]. 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 [46], social simulation [47] and software development [48, 49, 50, 51]. In competitive multi-agent systems, agents may detabe, negotiate and compete with each other in a game environment to achieve their goals, such as improving negotiation skills [52] and debating about the correct answer [53, 54, 55]. Some multi-agent systems may exhibit both cooperation and competition among agents. For example, WarAgent [56] models each country

[1] `http://apple-history.com/128k`

[2] `https://winworldpc.com/product/windows-3/31`

[3] `https://www.gnome.org/`

[4] `https://developer.android.com/studio`

[5] `https://developer.apple.com/xcode/`

[6] `https://cloud.google.com/sdk`

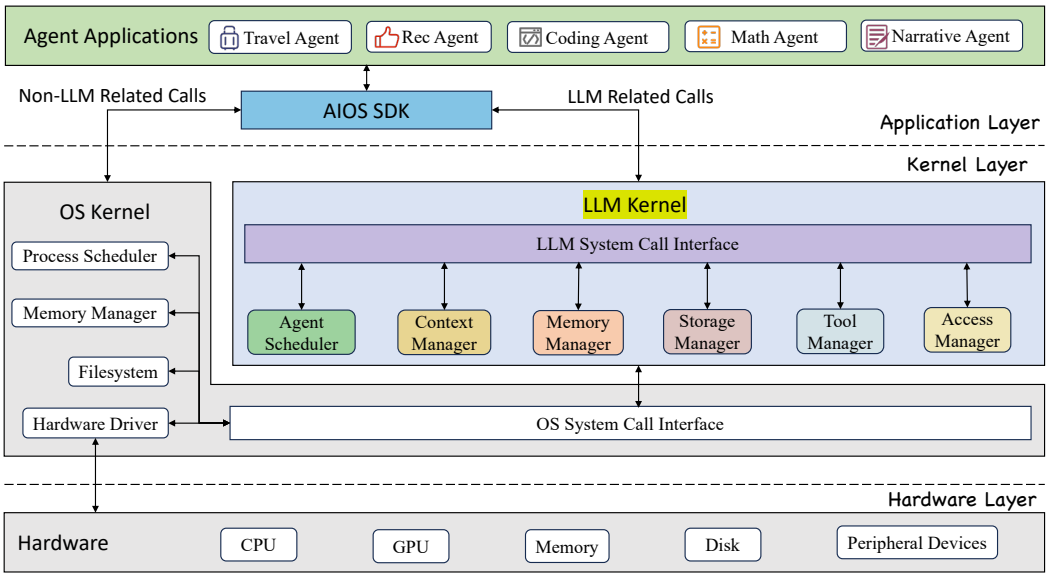

Figure 2: An overview of the AIOS architecture.

as an LLM-based agent to study how the interaction between countries can lead to international conflicts, where countries may cooperate with each other, such as establishing alliances and making peace agreements, or compete with each other, such as arms race, mobilization, and declaring wars.

3 AIOS Layers

As depicted in Figure 2, the architecture of our AIOS is organized into three distinct layers: the application layer, the kernel layer, and the hardware layer. This layered architecture ensures a clear delineation of responsibilities across the system. Each higher layer abstracts the complexities of the layers below it, facilitating interaction through interfaces or specific modules, thereby enhancing modularity and simplifying system interactions across different layers.

Application Layer. At the application layer, agent applications, such as travel agent or math agent, are developed and deployed. In this layer, AIOS provides the AIOS SDK, with a higher abstraction of system calls that simplifies the development process for agent developers. This SDK allows for development of agent applications by offering a rich toolkit that abstract away the complexities of lower-level system functions. This enables developers to dedicate their focus to the essential logic and functionalities of their agents, facilitating a more efficient development process.

Kernel Layer. The kernel layer is divided into two primary components: the OS Kernel and the LLM Kernel, each serving the unique requirements of non-LLM and LLM-specific operations, respectively. This distinction allows the LLM kernel to focus on LLM specific tasks such as context management and agent scheduling, which are essential for handling LLM-related activities and are not typically within the purview of standard OS kernel functions. Our work primarily concentrates on enhancing the LLM kernel without making significant alterations to the existing OS kernel structure. The LLM kernel is equipped with several key modules, including the LLM system call interface, agent scheduler, context manager, memory manager, storage manager, tool manager, and access manager. These components are designed to address the diverse execution needs of agent applications, ensuring efficient management and execution within the AIOS framework. The specifics of these modules will be further detailed in Section 4.

Hardware Layer. The hardware layer comprises the physical components of the system, including the CPU, GPU, memory, disk, and peripheral devices. It is crucial to note that the LLM kernel's system calls cannot directly interact with the hardware. Instead, these calls interface with the OS's system calls, which in turn manage the hardware resources. This indirect interaction ensures a layer of abstraction and security, allowing the LLM kernel to leverage hardware capabilities without requiring direct hardware management, thus maintaining the system's integrity and efficiency.

4 AIOS Implementation

In this section, we begin with an overview of the fundamental design and implementation of each module within the LLM kernel. Subsequently, we present the LLM system calls, which encompass essential functions for each module. At last, we discuss the exploration of the AIOS SDK, aiming to facilitate the development process for agent developers.

4.1 Agent Scheduler

Agent scheduler is designed to manage the agent requests in an efficient way. Consider the various agents (denoted as A, B, and C) in Figure 3, each of which has several execution steps. In the sequential execution paradigm, the agent tasks are processed in a linear order, where steps from a same agent will be processed first. This can lead to potential increased waiting times for tasks queued later in the sequence.

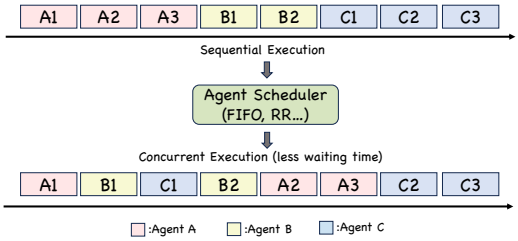

Figure 3: An illustration of the agent scheduler.

The agent scheduler employs strategies such as First-In-First-Out (FIFO)[7], Round Robin (RR)[8], and other scheduling algorithms to optimize this process. Through concurrent execution, the scheduler significantly balances waiting time and turnaround time of each agent, as tasks from different agents are interleaved and executed in parallel. This concurrent approach is visualized through a timeline where tasks from different agents are processed in an interleaved manner (e.g., A1, B1, C1, B2, A2, A3, C2, C3), ensuring that no single agent monopolizes the processing resources and that idle times are minimized. Apart from implementing the traditional scheduling algorithms, more complex scheduling algorithms considering the dependency relationships between agent requests can also be incorporated, which can be considered in the future.

4.2 Context Manager

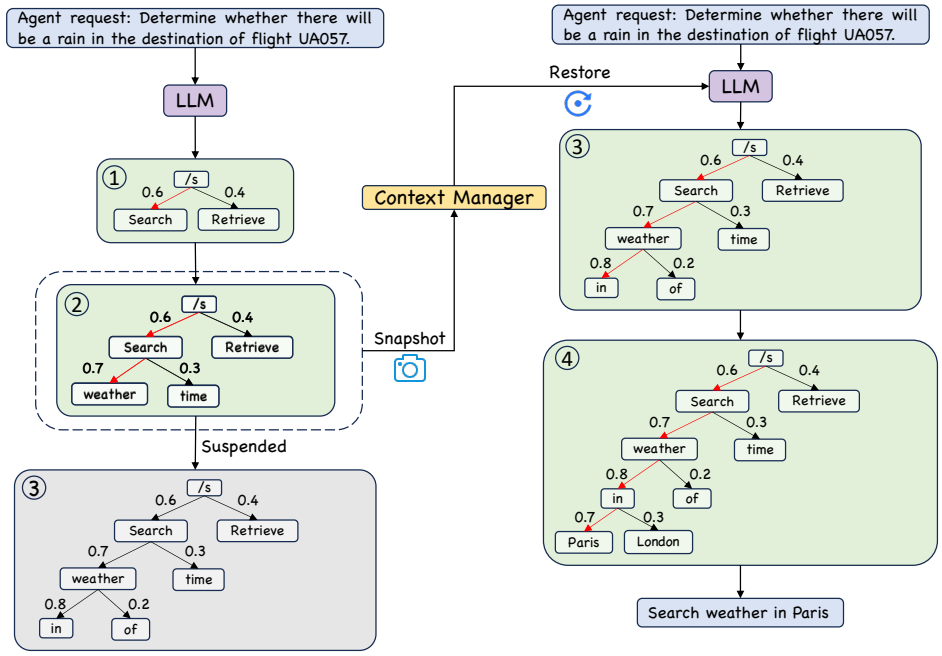

Figure 4: Context snapshot and restoration, where we use beam search (beam width = 1) as an example search algorithm to illustrate this generative decoding process.

[7] https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics)
[8] https://en.wikipedia.org/wiki/Round-robin_scheduling

The context manager is responsible for managing the context provided to LLM and the generation process given certain context. It primarily involves two crucial functions: context snapshot and restoration, and context window management.

Context Snapshot and Restoration. Consider that the scheduler algorithms may involve time quantum operations (e.g., Round-Robin) and agent requests may be suspended by the scheduler. This suspension happens even if the response has not been fully generated yet by the LLM. Therefore, it necessitates a mechanism to preserve the state of the LLM's generation process, ensuring that it can be accurately resumed once resources are available again.

AIOS provides the snapshot and restoration mechanisms in the context manager to address this issue, which can be seen from Figure 4. We use the beam search process[9], a typical practice in LLMs [10, 57, 58], to illustrate the generative decoding process. For simplicity of illustration, we set beam width as 1. Specifically, consider the agent request as: *Determine whether there will be a rain in the destination of flight UA057*. At each step, the LLM evaluates multiple potential candidates, with the most promising paths kept for further expansion based on the predefined beam width.

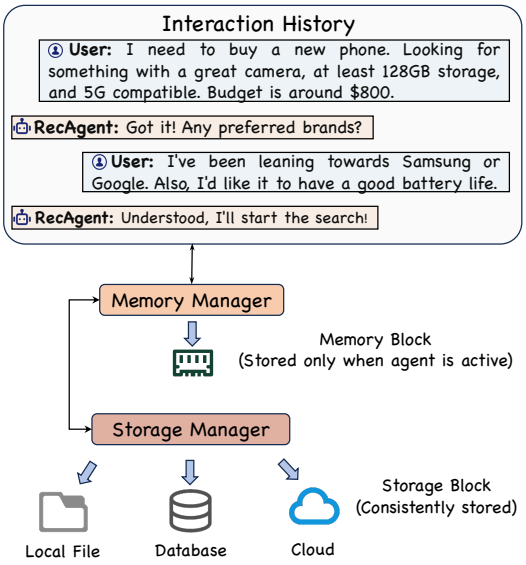

Figure 5: Storage of interaction history with memory manager and storage manager.

When such generation process has been suspended by the scheduler at an intermediate step, the context manager uses the snapshot function to capture and store the current state of the LLM's beam search tree, including all intermediate probabilities and paths being explored for generating the response. Upon resumption, the restoration function is employed to reload the saved state 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 optimizing resource use without compromising the quality and efficiency of response generation.

Context Window Management. To address challenges posed by long contexts that surpass the context window limit of LLMs, context manager also needs to manage potential expansion of context window. Specifically, context manager in AIOS supports basic text summarization and incorporates other expansion techniques [59, 60] to manage the context window. In this way, it can help enhance the LLM's ability to process and understand extensive contexts without compromising the integrity or relevance of the information.

4.3 Memory Manager

As shown in Figure 5, memory manager manages short-term memory within an agent's lifecycle, ensuring that data is stored and accessible only while the agent is active, either waiting for execution or during runtime. The current AIOS supports storing each agent's memory independently, each of which other agents have no direct access to, unless it is authorized by the access manager. More complicated memory mechanisms such as shared memory pools among agents or hierarchical caches can be considered and integrated into AIOS in the future. Compared with the storage manager introduced in the following, the memory manager enables rapid data retrieval and processing, facilitating swift responses to user queries and interactions without overburdening the storage of AIOS.

4.4 Storage Manager

In constrast, the storage manager is responsible for the long-term preservation of data, overseeing the storage of information that needs to be retained indefinitely, beyond the active lifespan of any

[9] `https://en.wikipedia.org/wiki/Beam_search`

Table 1: Managed tools in AIOS. The last column shows each tool's required input and output format.

Category	Tool Name	Description	Input → Output
Search	google_search	search information by Google API	Image/Text → Image/Text
	bing_search	search information by Bing API	Image/Text → Image/Text
Computation	currency_converter	currency converting	Text → Text
	wolframalpha	mathematical computation	Image/Text → Text
Database Query	sql_query	query data from SQL database	Text → Text
	wikipedia_query	query data from Wikipedia database	Text → Text
	arxiv_search	query articles from Arxiv	Text → Text
	words_api	query definitions and synonyms of English words	Text → Text
	urban_dictionary	query random words, the current word of the day	Text → Text
Image Processing	image_denoising	denoise images with noise	Image → Text
	image_deblurring	deblur images	Image → Text
	image_classification	classify images	Image → Text
	object_detection	detect objects in images	Image → Text
	face_rect	detect faces in images by FaceRect API	Image → Text

single agent. This permanent storage in AIOS is achieved through a variety of durable mediums such as local files, databases, or cloud-based solutions, ensuring data integrity and availability for future reference or analysis. The storage manager supports retrieval augmentation [61]. Through storing user preferences and maintaining historical interaction logs, the storage manager can enrich the agent knowledge update and enhancing the long-term user experience.

4.5 Tool Manager

The tool manager in the AIOS system manages a diverse array of API tools that enhance the functionality of LLMs. As shown in Table 1, the tool manager integrates commonly-used tools from various sources [7, 62, 63] and classify them into different categories, which covers web search, scientific computing, database retrieval, image processing, etc. In this way, the managed tools can cover different modalities of input and output (image and text), thus facilitating agent development within the AIOS ecosystem.

Table 2: Instances of LLM system calls.

Category	Name	Description	Arguments
Agent syscall	get_aid	get ID of an agent	-
	set_aid	set ID of an agent	int aid
	get_status	get status of an agent	int aid
	set_status	get status of an agent	int aid, string status
	get_priority	get priority of an agent	int aid
	set_priority	set priority of an agent	int aid, int priority
	suspend_agent	suspend an agent's operation	int aid
	resume_agent	resume an agent's operation	int aid
Context syscall	gen_snapshot	snapshot intermediate LLM generation status	-
	gen_restore	restore intermediate LLM generation status	-
	get_context	get context window length	-
	exp_context	expand context window length	-
	clr_context	clear current context window	-
	set_context	set a specific context window	context c
Memory syscall	mem_write	write the interaction record into memory	int aid, memory m
	mem_read	read the interaction record from memory	int aid, memory m
	mem_clear	clear specific memory record	int aid, memory m
	mem_alloc	allocate memory for an agent	int aid, size s
Storage syscall	sto_write	write the interaction record into storage	int aid, storage s
	sto_read	load the interaction record from storage	int aid, storage s
	sto_delete	delete the interaction record from storage	int aid, storage s
	sto_alloc	allocate storage for an agent	int aid, size s

4.6 Access Manager

The access manager orchestrates access control operations among distinct agents by administering a dedicated privilege group for each agent. Those other agents that are excluded from an agent's privilege group are denied access to its resources, such as the interaction history. To further enhance system transparency, the access manager compiles and maintains auditing logs. These logs capture

detailed information about access requests, agent activities, and any modifications to the access control parameters, which help to safeguard against potential privilege attacks [64, 65].

4.7 LLM System Call

LLM system call interface within the LLM kernel is designed to offer basic LLM call operation functions. This interface acts as a bridge between complex agent requests and the execution of different kernel's modules. As is shown in Table 2, analogous to OS system calls, LLM system calls offers a suite of basic functions that span across the kernel's modules, including agent management, context handling, memory and storage operations, and access control. The LLM system call list can be further expanded in the future to support more operations.

4.8 AIOS SDK

The AIOS SDK is designed to equip developers with a versatile toolkit for crafting sophisticated agent applications within the AIOS. This SDK encompasses a broad spectrum of functionalities, from initializing agents and managing agent lifecycles to facilitating complex operations like resource monitoring, and generation plan for an agent task. Like any operating system, enriching the SDK to be comprehensive and developer-friendly is a long-term and never-ending endeavor. The current SDK functions supported in AIOS are shown in Table 3, which will be continuously updated and expanded to fulfill the needs of evolving agent applications. This development effort aims to provide developers with the tools they need to harness the full potential of their agent applications within the AIOS framework.

Table 3: The list of SDK functions in AIOS SDK

SDK Function Name	Description	Return Type
`initializeAgent()`	set up environment for agent execution, including resource allocation	Void
`registerAgent()`	register a new agent to the system, providing it with ID and permissions	Boolean
`queueForExecution()`	enqueue the agent for execution	Void
`generatePlan()`	generate a plan of a given agent task	Object
`terminateAgent()`	terminate an agent's execution with cleanup	Void
`createAgentSnapshot()`	Create a snapshot of the agent's current state for rollback	Object
`rollbackAgentState()`	Rollback the agent's state to a previous snapshot	Void
`updateAgentConfig()`	update the configuration settings for an agent	Boolean
`authenticateAgent()`	authenticate an agent's credentials before execution	Boolean
`monitorResourceUsage()`	track and report the usage of system resources by agents	Object
`logEvent()`	provide logging of agent-specific events for debugging and auditing	Void
`listActiveAgents()`	list all currently active agents within the system	List
`queryAgentHistory()`	query the historical operations of an agent	Object
`encryptData()`	encrypt sensitive data generated by an agent	Object
`decryptData()`	decrypt sensitive data for an agent's consumption	Object

5 Evaluation

In this section, we evaluate both the correctness and the performance of AIOS modules when multiple agents are running in parallel in AIOS. Our investigation is guided by two research questions: firstly, whether the LLM responses to agent requests are consistent after agent suspension and transition to another agent, and secondly, how is the performance of AIOS scheduling in improving balance of waiting and turnaround time relative to non-scheduled (sequential) execution.

5.1 Setup

Our experiments are conducted in Python 3.9 with PyTorch 2.0.1 and CUDA 11.8 on an Ubuntu 22.04 machine equipped with 8 NVIDIA RTX A5000 GPUs. We employ the publicly available LLMs (i.e., Gemma-2b-it and Gemma-7b-it [66], LLaMA-2-13b-chat-hf [10]) as the backbone of AIOS. This selection is driven by the advantage of local deployment of open-sourced models, which aids in the accurate measurement of time latency. For the evaluation, we configure three specialized agents: a Math Agent for solving mathematical challenges, a Narrative Agent for generating novel narratives, and a Rec Agent tasked with providing restaurant recommendations. Each agent is designed to send 2 to 3 requests to the backbone LLM during running.

5.2 Experimental Results

Consistency Analysis. To answer the consistency question, we first run each of the three constructed agents individually to generate results. Subsequently, we execute these agents in parallel, capturing

Table 4: Consistency of LLM generated responses when running multiple agents in parallel compared with LLM generated responses when running a single agent one by one.

LLM-backbone	Math Agent		Narrative Agent		Rec Agent	
	BLEU Score	BERT Score	BLEU Score	BERT Score	BLEU Score	BERT Score
Gemma-2b-it	1.0	1.0	1.0	1.0	1.0	1.0
Gemma-7b-it	1.0	1.0	1.0	1.0	1.0	1.0
LLaMA-2-13b-chat-hf	1.0	1.0	1.0	1.0	1.0	1.0

their outputs at each step. To assess the consistency of outputs under the running of multiple agents in parallel and under the running of single agents one by one, we utilize the BLEU score [67] and BERT score [68] as evaluative metrics. Both metrics span from 0.0 to 1.0, with outputs produced in a single-agent context serving as the reference standard and we set the temperature parameter as 0 to eliminate the impact of randomness. As demonstrated in Table 4, BLEU and BERT scores all achieve the value of 1.0, indicating a perfect alignment between outputs generated in multi-agent and single-agent configurations. This result affirms the consistency of our design in effectively facilitating concurrent multi-agent operations.

Performance Analysis. To answer the efficiency question, we conduct a comparative analysis between AIOS employing FIFO scheduling and a non-scheduled approach, wherein the aforementioned three agents run concurrently. In the non-scheduled setting, the three agents are executed following a predefined sequential order: Math Agent, Narrative Agent, and Rec Agent. We employ two metrics for assessing temporal efficiency: waiting time (the interval from agent request submission to commencement) and turnaround time (the duration from agent request submission to completion). As each agent will send multiple requests to the LLM, the waiting time and turnaround time of each agent are calculated as the average of waiting time and turnaround time of all its sent requests, respectively. To mitigate randomness, we execute these three agents, both with and without scheduling, in five separate trials to report the results. As shown in Table 5, the non-scheduled approach exhibits good performance for agents earlier in the sequence, but at the expense of extended waiting time and turnaround time for agents later in the sequence. Conversely, AIOS's scheduling mechanism efficiently regulates both waiting time and turnaround time, a benefit that becomes particularly evident for agent requests submitted later by agents, especially when LLM is large. This suggests the importance of our scheduling to accommodate parallel operations of multiple agents.

Table 5: Effectiveness of agent scheduling, compared with non-scheduled (sequential) execution.

LLM backbone	Agent	Sequential execution (non-scheduled)		**Concurrent execution (scheduled)**	
		Waiting time (s)	Turnaround time (s)	Waiting time (s)	Turnaround time (s)
Gemma-2b-it	Math Agent	0.002±0.001	2.71±0.53	2.50±0.05	4.18±0.18
	Narrative Agent	2.18±0.53	3.18±0.64	3.34±0.17	4.18±0.18
	Rec Agent	4.78±0.96	7.68±1.27	3.46±0.20	5.91±0.19
Gemma-7b-it	Math Agent	0.002±0.001	4.95±0.10	5.92±0.01	10.75±0.19
	Narrative Agent	4.96±0.10	9.46±0.05	7.78±0.18	12.25±0.18
	Rec Agent	14.21±0.07	18.64±0.08	9.45±0.27	13.61±0.28
LLaMA2-13b-chat-hf	Math Agent	0.003±0.001	18.48±0.34	23.27±0.21	40.02±2.01
	Narrative Agent	18.49±0.34	35.89±0.76	28.61±2.10	46.15±2.02
	Rec Agent	53.86±1.15	71.62±1.26	34.92±2.22	52.44±2.43

6 Conclusions

This paper proposes the AIOS architecture, demonstrating the potential to facilitate the development and deployment of LLM-based agents, fostering a more cohesive, effective and efficient AIOS-Agent ecosystem. The insights and methodologies presented herein contribute to the ongoing discourse in both AI and system research, offering a viable solution to the integration challenges posed by the diverse landscape of AI Agents. Diverse future work can be built upon this foundation, exploring innovative ways to refine and expand the AIOS architecture to meet the evolving needs of developing and deploying LLM agents.

7 Future Work

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

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

Acknowledgement

We thank Jian Zhang, Zhenting Wang, and Wenyue Hua for their valuable discussions and suggestions during the project.

References

[1] Michael Wooldridge and Nicholas R Jennings. Intelligent agents: Theory and practice. *The knowledge engineering review*, 10(2):115–152, 1995.

[2] Nicholas R Jennings, Katia Sycara, and Michael Wooldridge. A roadmap of agent research and development. *Autonomous agents and multi-agent systems*, 1:7–38, 1998.

[3] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8:203–236, 2004.

[4] OpenAI. Gpt-4. `https://openai.com/research/gpt-4`, 2023.

[5] Facebook. Meta. introducing llama: A foundational, 65-billion-parameter large language model. `https://ai.facebook.com/blog/largelanguage-model-llama-meta-ai`, 2022.

[6] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[7] Yingqiang Ge, Wenyue Hua, Kai Mei, Juntao Tan, Shuyuan Xu, Zelong Li, and Yongfeng Zhang. OpenAGI: When LLM Meets Domain Experts. *Advances in Neural Information Processing Systems*, 36, 2023.

[8] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

[9] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*, 2022.

[10] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[11] Shijie Geng, Shuchang Liu, Zuohui Fu, Yingqiang Ge, and Yongfeng Zhang. Recommendation as language processing (rlp): A unified pretrain, personalized prompt & predict paradigm (p5). In *Proceedings of the 16th ACM Conference on Recommender Systems*, page 299–315, 2022.

[12] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

[13] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[14] Ross Taylor, Marcin Kardas, Guillem Cucurull, Thomas Scialom, Anthony Hartshorn, Elvis Saravia, Andrew Poulton, Viktor Kerkez, and Robert Stojnic. Galactica: A large language model for science. *arXiv preprint arXiv:2211.09085*, 2022.

[15] Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. Reasoning with language model is planning with world model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 8154–8173, 2023.

[16] Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks. *Advances in Neural Information Processing Systems*, 36, 2023.

[17] Steven I Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D Weisz. 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*, pages 491–514, 2023.

[18] Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, et al. Palm-e: an embodied multimodal language model. In *Proceedings of the 40th International Conference on Machine Learning*, pages 8469–8488, 2023.

[19] Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, et al. Do as i can, not as i say: Grounding language in robotic affordances. In *Conference on robot learning*, pages 287–318. PMLR, 2023.

[20] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. *International Conference on Learning Representations*, 2023.

[21] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2023.

[22] Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36, 2023.

[23] UW:CSE451. History of Operating Systems, 2023. https://courses.cs.washington.edu/courses/cse451/16wi/readings/lecture_readings/LCM_OperatingSystemsTimeline_Color_acd_newsize.pdf.

[24] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Commun. ACM*, 17(7):365–375, jul 1974.

[25] Charles Antony Richard Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

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

[27] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[28] Edsger W Dijkstra. Cooperating sequential processes. In *The origin of concurrent programming: from semaphores to remote procedure calls*, pages 65–138. Springer, 2002.

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

[30] Robert C Daley and Jack B Dennis. Virtual memory, processes, and sharing in multics. *Communications of the ACM*, 11(5):306–312, 1968.

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

[32] Marshall K McKusick, William N Joy, Samuel J Leffler, and Robert S Fabry. A fast file system for unix. *ACM Transactions on Computer Systems (TOCS)*, 2(3):181–197, 1984.

[33] Yingqiang Ge, Yujie Ren, Wenyue Hua, Shuyuan Xu, Juntao Tan, and Yongfeng Zhang. LLM as OS, Agents as Apps: Envisioning AIOS, Agents and the AIOS-Agent Ecosystem. *arXiv:2312.03815*, 2023.

[34] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

[35] Shunyu Yao and Karthik Narasimhan. Language agents in the digital world: Opportunities and risks. *princeton-nlp.github.io*, Jul 2023.

[36] Aaron Parisi, Yao Zhao, and Noah Fiedel. Talm: Tool augmented language models. *arXiv preprint arXiv:2205.12255*, 2022.

[37] Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, and Le Sun. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *arXiv preprint arXiv:2306.05301*, 2023.

[38] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. Webgpt: Browser-assisted question-answering with human feedback, 2022.

[39] Kechi Zhang, Ge Li, Jia Li, Zhuo Li, and Zhi Jin. Toolcoder: Teach code generation models to use apis with search tools. *arXiv preprint arXiv:2305.04032*, 2023.

[40] Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. Minedojo: Building open-ended embodied agents with internet-scale knowledge. *Advances in Neural Information Processing Systems*, 35:18343–18362, 2022.

[41] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. In *Intrinsically-Motivated and Open-Ended Learning Workshop@ NeurIPS2023*, 2023.

[42] Daniil A Boiko, Robert MacKnight, and Gabe Gomes. Emergent autonomous scientific research capabilities of large language models. *arXiv preprint arXiv:2304.05332*, 2023.

[43] Andres M Bran, Sam Cox, Andrew D White, and Philippe Schwaller. Chemcrow: Augmenting large-language models with chemistry tools. *arXiv preprint arXiv:2304.05376*, 2023.

[44] Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning*, pages 9118–9147. PMLR, 2022.

[45] Jiannan Xiang, Tianhua Tao, Yi Gu, Tianmin Shu, Zirui Wang, Zichao Yang, and Zhiting Hu. Language models meet world models: Embodied experiences enhance language models. *Advances in neural information processing systems*, 36, 2023.

[46] Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for "mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36, 2023.

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

[48] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2023.

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

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

[51] Martin Josifoski, Lars Klein, Maxime Peyrard, Yifei Li, Saibo Geng, Julian Paul Schnitzler, Yuxing Yao, Jiheng Wei, Debjit Paul, and Robert West. Flows: Building blocks of reasoning and collaborating ai. *arXiv preprint arXiv:2308.01285*, 2023.

[52] Yao Fu, Hao Peng, Tushar Khot, and Mirella Lapata. Improving language model negotiation with self-play and in-context learning from ai feedback. *arXiv preprint arXiv:2305.10142*, 2023.

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

[54] Chi-Min Chan, Weize Chen, Yusheng Su, Jianxuan Yu, Wei Xue, Shanghang Zhang, Jie Fu, and Zhiyuan Liu. Chateval: Towards better llm-based evaluators through multi-agent debate. In *The Twelfth International Conference on Learning Representations*, 2023.

[55] Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Zhaopeng Tu, and Shuming Shi. Encouraging divergent thinking in large language models through multi-agent debate. *arXiv preprint arXiv:2305.19118*, 2023.

[56] Wenyue Hua, Lizhou Fan, Lingyao Li, Kai Mei, Jianchao Ji, Yingqiang Ge, Libby Hemphill, and Yongfeng Zhang. War and peace (waragent): Large language model-based multi-agent simulation of world wars. *arXiv preprint arXiv:2311.17227*, 2023.

[57] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[58] Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*, pages 2397–2430. PMLR, 2023.

[59] Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*, 2023.

[60] Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. Yarn: Efficient context window extension of large language models. *arXiv preprint arXiv:2309.00071*, 2023.

[61] Grégoire Mialon, Roberto Dessi, Maria Lomeli, Christoforos Nalmpantis, Ramakanth Pasunuru, Roberta Raileanu, Baptiste Roziere, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, et al. Augmented language models: a survey. *Transactions on Machine Learning Research*, 2023.

[62] LangChain. Langchain. `https://github.com/langchain-ai/langchain`, 2024.

[63] Rapid. Rapid api hub. `https://rapidapi.com/hub`, 2024.

[64] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.

[65] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on android. In *NDSS*, volume 17, page 19, 2012.

[66] Tris Warkentin Jeanine Banks. Gemma: Introducing new state-of-the-art open models. `https://blog.google/technology/developers/gemma-open-models/`, 2024.

[67] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[68] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*, 2019.