AI METROPOLIS: SCALING LARGE LANGUAGE MODEL-BASED MULTI-AGENT SIMULATION WITH OUT-OF-ORDER EXECUTION

Zhiqiang Xie [1] **Hao Kang** [2] **Ying Sheng** [1] **Tushar Krishna** [2] **Kayvon Fatahalian** [1] **Christos Kozyrakis** [1]

ABSTRACT

With more advanced natural language understanding and reasoning capabilities, agents powered by large language models (LLMs) are increasingly developed in simulated environments to perform complex tasks, interact with other agents, and exhibit emerging behaviors relevant to social science research and innovative gameplay development. However, current multi-agent simulations frequently suffer from inefficiencies due to the limited parallelism caused by false dependencies, resulting in a performance bottleneck. In this paper, we introduce AI Metropolis, a simulation engine that improves the efficiency of LLM agent simulations by incorporating out-of-order execution scheduling. By dynamically tracking real dependencies between agents, AI Metropolis minimizes false dependencies, enhances parallelism, and maximizes hardware utilization. Our evaluations demonstrate that AI Metropolis achieves speedups from $1.3\times$ to $4.15\times$ over standard parallel simulation with global synchronization, approaching optimal performance as the number of agents increases.

1 INTRODUCTION

Large Language Models (LLMs) are advanced machine learning models trained on vast amounts of data, excelling in understanding and generating natural language. They have transformed natural language processing, enabling high-accuracy applications like text completion (Merity et al., 2016), summarization (Narayan et al., 2018), and reasoning (Cobbe et al., 2021). Beyond simple queries and chatbot interactions (OpenAI, 2024a), there is growing interest in using LLMs to create self-planning, decision-making, problem-solving, and reasoning engines (Wang et al., 2024). These advancements aim to develop human-like agents (Xi et al., 2023) capable of performing complex tasks, interacting with environments and other agents, and making informed decisions based on context.

This interest is particularly pronounced in developing LLM-powered agents within simulated environments, where two unique opportunities arise. First, simulation environments provide an efficient platform for testing and tuning LLM agents (Dubois et al., 2024; Liu et al., 2023; Wang et al., 2023b), with potential applications extending to real-world settings or virtual environments like gaming. Second, the enhanced natural language understanding and reasoning capabilities of LLMs have sparked a trend of examining emergent social behaviors of these agents in game-like simulations (Park et al., 2023; Altera.AL et al., 2024). Such studies can serve as predictive models, forecasting real-world human behaviors, which is highly valuable for social science research (Ziems et al., 2023; Grossmann et al., 2023).

Despite the significance of simulation environments for LLM agents, the efficiency of managing simulation states and scheduling LLM requests in simulations are often overlooked, leading to slow and inefficient simulation processes. Recent research commonly implements their LLM agents simulation (Park et al., 2023; Gong et al., 2023) directly adhering to a paradigm borrowed from reinforcement learning agent training and traditional multi-agent simulation (Emau et al., 2011), where simulation time is discretized into time steps and a *step* (or similar) function is invoked to apply agents' actions, synchronize the environment, and coordinate agents at each interval. This pattern, illustrated in Algorithm 1, is prevalent in prominent reinforcement learning frameworks such as OpenAI Gym (Brockman et al., 2016), Meta Pearl (Zhu et al., 2024), and TensorFlow Agents (Guadarrama et al., 2018). The rationale behind this design is that global synchronization, enforced through the *step* function, easily maintains *temporal causality* within the simulation by serializing tasks along the simulation time axis.

While this design suits the needs of reinforcement learning and traditional multi-agent simulations, we found it inefficient for LLM agents due to their unique performance characteristics, necessitating a novel scheduling approach. Simulations involving LLM agents, like other LLM-powered applications, are heavily dominated by inference time. Tak-

[1]Stanford University [2]Georgia Institute of Technology. Correspondence to: Zhiqiang Xie <xiezhq@cs.stanford.edu>.

Proceedings of the 8^{th} MLSys Conference, Santa Clara, CA, USA, 2025. Copyright 2025 by the author(s).

ing the pioneering work on generative agents (Park, 2024) (GenAgent) as an example, our trace analysis reveals that approximately 95% of the simulation time is dedicated to LLM inference. Consequently, inference throughput becomes crucial, as higher throughput directly translates to shorter completion times and lower costs. Furthermore, recent studies on LLM serving engines (Kwon et al., 2023; Zheng et al., 2024) indicate that large batch sizes are essential for achieving high inference throughput. Unfortunately, the traditional approach, which enforces step-wise temporal causality across simulation steps, introduces excessive synchronization that significantly reduces parallelism, thereby reducing achievable batch sizes and leading to low throughput. This reduction in parallelism occurs because the execution times of LLM queries from different agents within a simulation step can vary significantly due to two main reasons: (1) variations in the input and output lengths of queries, and (2) differing numbers of queries sent by different agents. As a result, enforcing global synchronization causes many agents to wait unnecessarily for each step to complete, limiting concurrent LLM queries and further reducing throughput. This reduction in parallelism also hampers scalability, as adding more resources fails to meaningfully decrease the overall simulation completion time.

Our key observation is that temporal causality in simulations can be maintained without the costly global step-wise synchronization in the simulation. Intuitively, if two agents are far apart in a simulated world, the actions of one agent will not be immediately visible to the other. This means that it is often unnecessary for all agents to wait for each step to finish before proceeding, revealing a false dependency that can be removed to improve efficiency in the simulation.

To address the aforementioned challenge, in this paper, we present AI Metropolis, a multi-agent simulation engine for LLM-powered agents that introduces the concept of out-of-order execution to simulation scheduling. By carefully tracking real dependencies between agents during runtime, we can effectively eliminate most false dependencies. This approach allows certain agents to advance in simulation time ahead of others without affecting the simulation's outcome, which significantly enhances parallelism and thus better utilizes hardware with larger inference batch sizes. Dependency tracking is achieved by analyzing the temporal-spatial relationships between agents, where the number of steps an agent can advance is determined by its distance from other agents. Similar to the scoreboard in out-of-order execution algorithms, AI Metropolis maintains a specialized dependency graph to efficiently track these relationships.

AI Metropolis provides LLM agent developers with interfaces similar to OpenAI Gym (Brockman et al., 2016), while seamlessly managing simulation state updates, database I/O, scheduling, and LLM inference processes. We evaluated AI Metropolis by replaying traces collected from instrumenting the original GenAgent implementation (Park, 2024) across different models, GPUs, and simulation scales. The results demonstrate that AI Metropolis outperforms the standard approach of parallel simulation with global step synchronization, achieving speedups from $1.3\times$ to $4.15\times$, and approaching an order of magnitude improvement over the original GenAgent implementation. As the number of agents increases, AI Metropolis rapidly nears optimal performance, demonstrating its scalability and effective dependency management. We have open-sourced AI Metropolis to accelerate research in large-scale LLM agent simulation. In addition, we released the collected GenAgent traces to address a critical gap in LLM serving benchmarks, particularly in capturing the unique and complex dependency patterns among LLM calls. Both are available at: `https://github.com/xiezhq-hermann/ai-metropolis`.

Algorithm 1 Traditional Simulation Scheduling

```
Input: target_step, agents, world
Initialize: step ← 0
while step < target_step do
    actions ← [ ]
    for all agent in agents do
        actions.append(agent.proceed(world)) [a]
    end for
    world.step(actions)
    step ← step + 1
end while
```

[a] The ***proceed*** function involves LLM calls to process tasks.

2 Simulation of LLM Agent Interaction

2.1 Background

While simulation environments vary greatly, featuring diverse and complex action spaces and interactions, they typically adhere to a common high-level procedure outlined in Algorithm 1. In this procedure, agents determine their next actions based on the current state of the world and their internal states. These actions, in turn, update the world state as the simulation progresses, subsequently influencing future agent behaviors. The functions *agent.proceed* and *world.step* are provided by agent and environment developers, and can be implemented through manually defined rules, calls to LLMs, or a combination of both.

To illustrate the challenge of achieving efficient simulation, we use GenAgent as a concrete example within the broad family of simulations for LLM agent interaction. GenAgent proposes a comprehensive agent architecture and interaction mechanisms that have inspired substantial subsequent

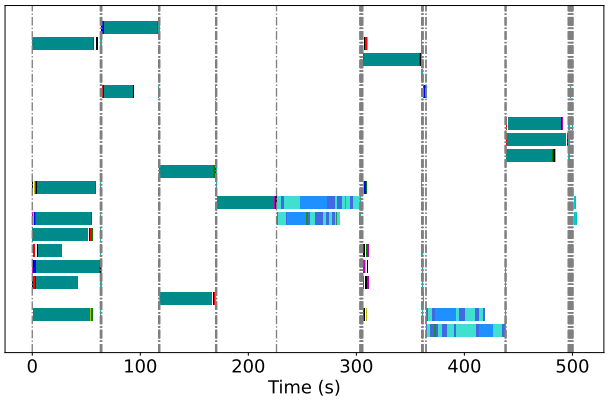


Figure 1. A snippet of the execution trace of a GenAgent simulation. The x-axis shows the elapsed execution time, with each row representing an agent's stream of LLM invocations. Colored bars denote different agent functions, and black dashed vertical lines indicate the completion of each step.

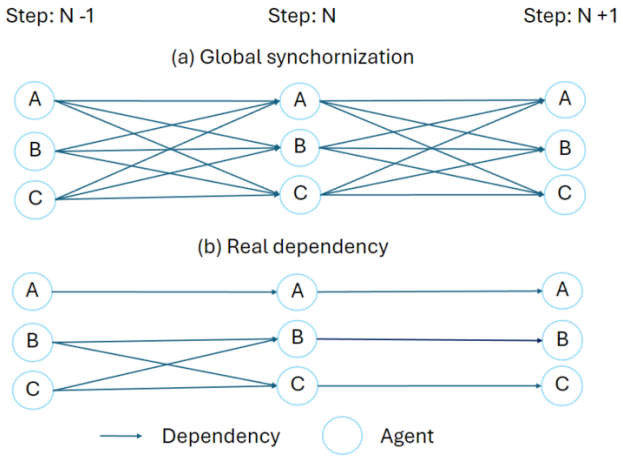


Figure 2. The dependency between agents' tasks is introduced by temporal causality. The top illustration shows an overly strict enforcement of this dependency, while the bottom illustration depicts a case of actual dependency.

research. Their concepts and workflows are widely adopted in the community. In GenAgent, 25 agents inhabit a world called SmallVille, akin to a grid-based game. Each agent possesses its own personality, social relationships, and daily routines. They navigate the world, interact with objects, and converse with other agents. The *agent.proceed* function on line 6 of Algorithm 1 is expanded to Algorithm 2 to detail several steps: perceiving surroundings, planning actions based on recent events, recalling relevant past events from memory, following a structured daily routine, and occasionally reflecting on their actions or experiences. Each of these steps can involve LLM calls. Each step corresponds to ten seconds in the simulated time in GenAgent.

Algorithm 2 Proceed function in GenAgent

1: **Input:** agent, world
2: **Output:** action
3: perceived_events ← agent.***perceive***(world)
4: retrieved_events ← agent.***retrieve***(perceived_events)
5: action ← agent.***plan***(world, retrieved_events)

2.2 Motivation and Challenges

Imbalanced workload reduces available parallelism. Although line 5 in Algorithm 1 suggests that parallelism can scale up to the number of agents, the effective parallelism is often significantly lower due to workload imbalance among agents. As illustrated in Figure 1, there are moments when many agents send out LLM requests uniformly. However, for the majority of the execution time, a few agents dominate each step, resulting in prolonged idle periods for many other agents who issue no LLM queries. This sparsity inherently arises from agents having independent schedules and encountering distinct events, making even distributions unlikely. Additionally, as shown in Figure 1, even for the same type of LLM calls, completions can vary significantly depending on inputs provided by different agents, further introducing imbalance. Our measurements indicate that for a full-day simulation of 25 agents, there are, on average, only 1.94 concurrent LLM queries throughout the simulation.

False Dependency. While the workload imbalance across agents might be inevitable, low parallelism is not. We found that the primary cause of idleness is the overly strict enforcement of time causality. Requiring all events in one time step to complete before advancing to the next step introduces unnecessary dependencies. As shown in Figure 2, this approach creates an implicit all-to-all dependency across agents in consecutive steps. However, some agents, such as agent A, may be sufficiently isolated and unable to interact with others, thus not creating dependencies on agents like B or C. Our trace analysis for a whole day simulation of GenAgent indicates that, on average, each agent is dependent on only 1.85 agents (including itself) from the prior step, far less than the default 25. The issue of false dependencies worsens as the agent count grows, as more false dependencies are enforced, diminishing the benefits of increased parallelism. Although agents' behaviors are driven by responses from LLMs, which limits a scheduler's ability to optimally manage dependencies without foresight, agent behavior is still somewhat predictable. Agents are constrained by their movement speed and limited action space, providing an opportunity to reduce most false dependencies through analysis of agents' temporal-spatial relationships.

Requests of Different Priorities. The dependency lens also reveals something unique about simulation compared to common LLM services like chatbots: there are long critical paths in the task of simulation, consisting of a chain of LLM calls that cannot be parallelized. Therefore, requests have different priorities; those on the critical path should be served before non-critical requests to minimize the overall completion time as much as possible.

3 DESIGN OF AI METROPOLIS

Motivated by observations described in §2.2, we design AI Metropolis, an optimized simulation engine that serves as middleware between the developer-defined world and agents and the LLM serving engine, efficiently managing state updates and scheduling LLM queries. By allowing agents to progress at varying speeds based on their LLM call loads, AI Metropolis eliminates the need for frequent global synchronization, reducing false dependencies and maximizing parallelism. Algorithm 3 provides an overview of the new scheduling workflow adopted by AI Metropolis, contrasting it with the traditional time step synchronized scheduling shown in Algorithm 1.

Algorithm 3 AI Metropolis Scheduling Workflow

```
Input: target_step, agents, world
Initialize: base_step ← 0, ready_agents ← agents
worker_pool ← InitProcessPool(process_routine)
ready_queue ← PriorityQueue()
ack_queue ← PriorityQueue()
dependency_graph ← Graph(agents)
{Controller}
while base_step < target_step do
    ready_clusters ← geo_clustering(ready_agents)
    ready_queue.put(ready_clusters)
    ack_cluster ← ack_queue.get()
    ready_agents ← update_agents(agents, ack_cluster)
    base_step ← update_base_step(ack_cluster)
end while
{Worker}
while base_step < target_step do
    cluster ← ready_queue.get()
    actions ← [ ]
    for all agent in cluster do
        actions.append(agent.proceed(world))
    end for
    world.resolve_conflict_and_commit(actions)
    dependency_graph.update(cluster)
end while
```

3.1 Overview

Below, we define essential terms used in AI Metropolis:

- **Blocked**: An agent A becomes *blocked* if it has to wait for another agent B to finish the current step before it can proceed, ensuring temporal causality and simulation correctness. This is formally defined in §3.2.
- **Coupled:** Agents A and B, become *coupled* if they are sufficiently close, interact with each other, and thus must proceed together. This is formally defined in §3.2.
- **Cluster**: A cluster is a group of *coupled* agents at the same step. Each agent can independently issue requests to LLMs within its *proceed* function. However, the entire group needs to synchronize at the end of the step to resolve potential conflicts and avoid dependency violations as described in §3.2.
- **Worker**: A worker is a process that handles one cluster, to proceed one step at a time. Within the worker process, each agent in the cluster operates in its own thread to communicate with the LLM serving engine and process its tasks. Workers are independent processes without synchronization between them, and the number of workers can be adjusted based on available CPU resources.
- **Controller**: The controller is the main process of the simulation engine. After initializing the world, it periodically communicates with workers through two queues: it prepares tasks for workers via the *ready_queue* and confirms the completion of clusters from workers through the *ack_queue*.

During a simulation, workers continuously pull clusters from the *ready_queue*. After finishing the step for a cluster, a worker updates the dependency graph stored in a database for all agents in the cluster and then places the completed cluster in the *ack_queue* as a completion confirmation. Simultaneously, the controller continuously pulls notifications from the *ack_queue*. Each time it processes a confirmation, it queries the dependency graph to filter out the agents that are not *blocked*, creating new ready clusters out of the ready agents and placing them into the *ready_queue* promptly. This workflow allows agents to advance steps ahead of others as long as dependency permits. Notably, both the *ready_queue* and *ack_queue* are priority queues that automatically sort tasks based on their associated steps.

We discuss the the dependency tracking mechanism in §3.2, the spatiotemporal dependency graph realizing the mechanism in §3.3, agent clustering in §3.4, priority scheduling in §3.5, and key implementation details in §3.6.

3.2 Dependency Tracking

As described in §2.2 and shown in Figure 2, temporal causality introduces the dependence of tasks in a simulation. Using

the language of computer systems, temporal causality creates an order of a set of reads and writes on a shared memory. At the beginning of each step, agents read different parts of the environment and at the end, they commit writes to different parts. Therefore, the tasks of an agent across different steps must be serialized, as it reads what it wrote in the last step. The dependency between tasks of different agents can then also be formulated as a read-after-write data dependency. For two agents A and B, if A is about to observe a part of the world at step $Step_A$, and B is about to write to that part of the world at step $Step_B$. if $Step_B$ is smaller than $Step_A$, meaning that the write is designed to happen before the read, then A must wait for B to complete step $Step_B$ before starting the tasks in step $Step_A$.

In simulations, each agent typically perceives only a portion of the world, defined by the surrounding area within a specified radius, denoted as $radius\_p$. We also assume a maximum speed limit, denoted as max_vel, governing both agents' movement and information propagation within each step. For instance, in GenAgent, an agent perceives an area within a radius of 4 grid units and can modify the status of an adjacent grid by interacting with an object or agent on it or moving to it. This defines the region the agent can read from and write to. Consequently, to maintain read-after-write data dependency, the following condition must hold throughout the simulation:

$$\forall \text{ agents } A, B, \text{ and their current steps } Step_A \text{ and } Step_B,$$
$$\text{if } Step_A \neq Step_B, \text{then } dist(A, B) > radius\_p$$
$$+ (|Step_A - Step_B| - 1) \times max\_vel$$

where $dist(A, B)$ denotes the distance between A and B. This ensures a read in a later step never overlaps with regions potentially influenced by writes from previous steps. Intuitively, this means that agents never perceive other agents who exist at different times. To enforce this condition, we can derive the following rules to determine the relationships between the tasks of agents. The complete derivation can be found in Appendix A and the following are the rules AI Metropolis uses: for any two agents A and B and their tasks in steps $Step_A$ and $Step_B$,

- We define that A and B are *coupled* if $Step_A = Step_B$ and $dist(A, B) \leq radius\_p + max\_vel$, which means they must be grouped into the same cluster and proceed to the next step together.
- We define that A is *blocked* by B if $dist(A, B) \leq (Step_A - Step_B + 1) \times max\_vel + radius\_p$, meaning that A cannot begin tasks in step $Step_A$ until B advances to the next step.
- A cluster can freely advance to the next step if none of its agents are blocked by any other agent.

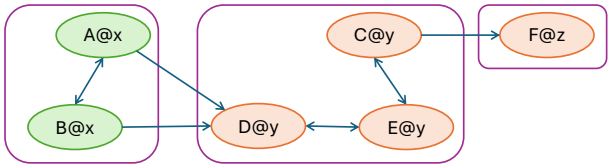


Figure 3. An example of a spatiotemporal dependency graph. Each node, such as A@x, represents an agent (A) at a specific time step (x). Single arrows indicate dependencies, while double arrows represent coupled relationships between agents. Purple boxes denote clusters of agents, where green nodes indicate agents that are free to proceed and orange nodes represent blocked agents.

Note that this set of rules are conservative, meaning the read in the later step will wait for all potential writes to a certain region to finish, even if the write does not occur eventually. While this may still preserve some false dependencies, we show in §4 that it achieves performance close to the optimal. Additionally, this design does not require a data race detector and correction mechanism, making it more scalable and easier to implement.

3.3 Spatiotemporal Dependency Graph

AI Metropolis uses rules defined in §3.2 to construct a special graph that tracks dependencies between agents. It is named spatiotemporal dependency graph and each node in the graph represents an agent, containing its temporal (time step) and spatial (coordinates on the map) information. An edge $A \rightarrow B$ indicates that agent B is currently *blocked* by agent A, while $A \leftrightarrow B$ signifies that agents A and B are coupled into a cluster as shown in Figure 3.

AI Metropolis maintains this graph in an in-memory database. Whenever a worker advances a cluster to the next step, it re-examines relationships between each agent in the cluster and other relevant agents, applying predefined rules. Any resulting changes in relationships are then recorded in the database. An efficient coordinate-based geo-query approach is employed to confine each agent's checks to neighbors within a specific radius, resulting in an effective common-case complexity of $O(S \times N)$. To further minimize overhead, the update algorithm is implemented in a parallel manner with transactional update enforcement to prevent data races. As a result, a typical local graph update for a cluster of agents takes about 1 ms, accounting for less than 1% of total runtime. Afterward, the worker process places a completion confirmation into the *ack_queue*.

The dependency graph will be utilized by the controller process for efficiently identifying the agents that are not *blocked*, allowing it to release maximum parallelism.

3.4 Agent Clustering

When agents are close enough to each other, they perceive each other's actions committed in the last step. In other words, they collectively read what they wrote in the last step. They might also encounter write conflicts that must be resolved by developer-specified rules; for example, two agents might both want to use the bathroom, but only one can step in. These potential interactions couple them into a cluster that must proceed together. Whenever there are new ready agents who are not blocked, the controller process runs *geo_clustering* to group coupled agents into clusters based on the rule described in §3.2. If none of the members of a cluster are blocked, the cluster is considered ready and will be placed into the *ready_queue*. Using clusters as the minimal synchronized units, as opposed to synchronizing all agents as described in Algorithm 1, effectively reduces false dependencies and scheduling overhead.

3.5 Priority Scheduling

As motivated in §2.2, allowing agents to process tasks associated with different time steps simultaneously creates requests of varying priorities. We found that the time step associated with a request serves as a good measure of its priority. A write operation in a prior step can block many reads in subsequent steps; intuitively, the smaller the time step, the more future actions it can potentially block. To enhance parallelism, we maintain both the *ready_queue* and *ack_queue* as priority queues, prioritizing the execution of tasks from earlier steps. No preemption during LLM inference is applied as that might cause extra overhead (Sheng et al., 2023b). We demonstrate the effectiveness of this priority scheduling in §4.4.

3.6 System Implementation

In addition to the design choices around dependency management, clustering, and priority scheduling that enable AI Metropolis to expose more parallelism from the simulation, it is also worth highlighting some of the design choices that make AI Metropolis scalable:

Proper Mapping of Parallelism. Choosing the right programming abstraction for different tasks is critical to scalability. AI Metropolis employs threads for agents, as the need for synchronization within clusters requires low-overhead communication. Processes manage the controller and workers to bypass Python's Global Interpreter Lock (GIL) and facilitate scaling beyond a single machine.

Light and Fast Critical Path on the Controller Process. All critical path tasks are implemented in C++ to minimize overhead and circumvent Python's GIL limitations. Furthermore, heavy lifting, including complex agent processing logic and dependency graph updates, is offloaded to concurrent workers. This approach lightens the critical path for the controller process, minimizing workers' waiting time for task allocation.

Scalable I/O. Except for *ready_queue* and *ack_queue*, all inter-process synchronization are handled through an in-memory (Redis) database. This database also handles transactional updates for all simulation states and stores instrumentation data, supporting automatic scalability beyond a single node.

Decoupling Simulation Engine from LLM Serving Engine. In AI Metropolis, only workers communicate with the LLM serving engine through a thin shim layer, providing easy observability and scalability.

We implemented the core of AI Metropolis in about 1k lines of C++ and 2k lines of Python code. An additional 3k lines of Python code (excluding assets and prompts) were written to port the GenAgent simulation using our interfaces. This is about 50% of the code compared to the original implementation, achieving up to an order of magnitude speed-up and promising scalability, as demonstrated in §4.

4 Evaluation

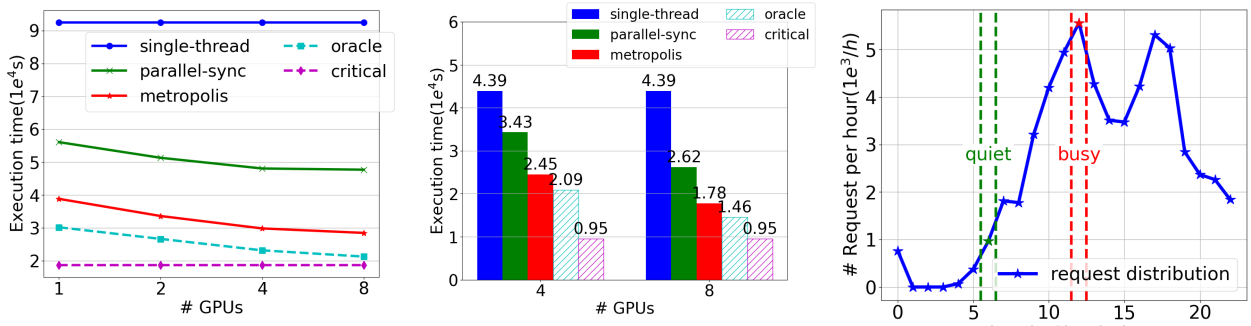
In the evaluation, we aim to answer the following questions:

- Does AI Metropolis effectively enhance parallelism by tracking real dependencies, and how does this translate to shorter completion times?
- Does AI Metropolis scale as the size of the simulated world increases and the number of agents grows?
- Given that AI Metropolis does not eliminate all false dependencies as illustrated in §3.2, how well does it perform compared to the optimal solution?

We describe the experimental setup in §4.1 and discuss the performance results of full-day simulations at a small scale in §4.2, which uses the same simulation settings reported in the GenAgent paper. We then examine the performance comparisons as the size of the world increases and the number of agents grows to a thousand, assessing scalability in §4.3. Finally, we conduct a performance breakdown in §4.4 to demonstrate the effectiveness of priority scheduling.

4.1 Methodology

Serving Engine. We use SGLang (Zheng et al., 2024) (v0.1.17) as the LLM serving engine, as it is not only one of the state-of-the-art LLM serving engines but also lightweight and easy to instrument and modify. For consistent and stable performance benchmark results, we turned



(a) Simulation using Llama-3-8b-instruct on NVIDIA L4 GPU

(b) Simulation using Llama-3-70b-instruct on NVIDIA A100 GPUs

(c) LLM query distribution in simulation

Figure 4. (4a, 4b) End-to-end 25 agents full day simulation completion time with different number of GPUs. (4c) shows the distribution of LLM calls over the simulated hours, note the low activity period during 1 a.m. - 4 a.m. is because all agents are sleeping.

off its automatic common prefix caching feature; however, enabling the cache generally provides about a 20% throughput gain across all settings.

Model and Hardware Platform. We benchmarked AI Metropolis with various models and GPUs to assess its effectiveness across different sizes and complexities. We chose state-of-the-art open-source LLMs from the Meta Llama-3 instruct series (Meta, 2024). Community benchmarks (Chiang et al., 2024) indicate that the smallest 8B model already surpasses the ChatGPT-3.5 model used in the original GenAgent paper, making it ideal for performance evaluation. We benchmarked our system with both the 8B and 70B models. The 8B model offers a lightweight deployment option, while the 70B model provides advanced capabilities, though at a higher cost. For the Llama-3 8B experiments, we used NVIDIA L4 GPUs on GCP G2 instances, scaling from one to eight GPUs to assess data parallelism. For the Llama-3 70B experiments, we used NVIDIA A100-80GB GPUs, applying tensor parallelism across four GPUs, and expanding to eight GPUs for a hybrid data and tensor parallelism configuration. Additionally, we benchmarked AI Metropolis using the Mixtral-8 $\times$ 7B-Instruct-v0.1 (Mistral AI, 2023) model, a mixture of expert models, on the same A100 platform which can leverage higher data parallelism to reveal more performance characteristics.

Traces. We collected workload traces for 40 simulation days of GenAgent by instrumenting the original implementation (Park, 2024) and running it multiple times using the same settings reported in the paper. OpenAI GPT-3.5 API service (OpenAI, 2024b) was used as the LLM engine as the same setting in the paper. On average, each simulation day's trace consists of $56.7k$ LLM call events. Each event includes the input prompt, configurations, LLM response, calling step, and caller's identity. A separate trace file tracks the agent's movements throughout the simulation. The average length of input tokens is 642.6, and the average length of output tokens is 21.9. We conducted the performance benchmark using the replay mode of AI Metropolis, faithfully replaying these traces to ensure the same movements, interaction patterns, inputs, and the same length of generation output by setting *ignore_eos* in SGLang for comparable and stable performance results.

4.2 Full Day Simulation in SmallVille

We benchmark AI Metropolis using the same setup described in the GenAgent paper, which involves 25 agents within a world named SmallVille, a 100×140 grid, running for a full simulation day.

The following experiment settings are used in benchmark:

- *single-thread* employs a single thread to handle simulation states and issue LLM requests, as per the design adopted by the original implementation to simplify simulator implementation. No parallelism is exposed for LLM requests from different agents.
- *parallel-sync* is a stronger baseline in which all agents operate within the same time step can independently issue LLM requests. Although global synchronization inherently limits achievable parallelism, it represents a standard and effective strategy for synchronous multi-agent simulations, as discussed in § 2.1. We implemented this baseline as a mode of AI Metropolis.
- *oracle* represents the optimal dependency management solution for comparison. This setting constructs an optimal dependency graph by analyzing the full trace and mining all necessary dependencies based on agent interactions. For example, if two agents appear in each other's observation space, they synchronize before and after the step to ensure temporal causality. This setting is unattainable in real systems and serves to illustrate

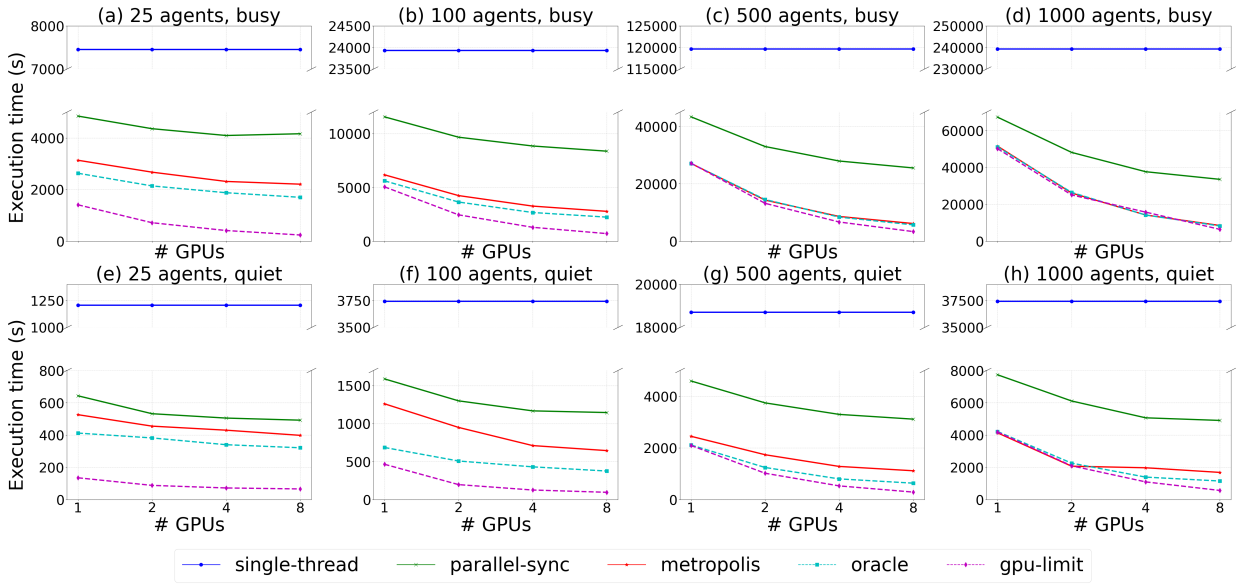


Figure 5. Benchmark of busy (12 p.m. - 1 p.m.) and quiet (6 a.m. - 7 a.m.) hours using Llama-3-8b-instruct on NVIDIA L4 GPUs, with agent counts scaled from 25 to 1000. Single-thread results for 500 and 1000 agents are projected based on workload estimations.

the potential improvement of dependency management. By having an optimal dependency graph, the most available parallelism will be released.

- *critical* refers to the critical path of the simulation, extracted from the optimal dependencies used in the oracle setting. It identifies the path containing the most LLM input and output tokens, setting a lower bound of completion time regardless of available resources.

First, AI Metropolis outperforms the *single-thread* and *parallel-sync* baselines by $2.38\times$ and $1.44\times$ on a single L4 GPU. As additional GPUs are employed, increasing the demand for parallelism, the speedup rises to $3.25\times$ and $1.67\times$ respectively on 8 GPUs. We also measured the achieved parallelism for each simulation by averaging the number of outstanding requests over the execution time, where AI Metropolis reached 3.46, compared to 0.95 for *single-thread* and 1.94 for *parallel-sync* on 8 GPUs. These results align with the observed speedups, as greater parallelism improves GPU utilization and overall performance.

AI Metropolis also approaches *oracle* performance, reaching 74.7% of the oracle performance on 8 GPUs and up to 82.9% on a single GPU. This gap arises from AI Metropolis's longer critical path compared to the oracle baseline, as it conservatively prevents certain agents from advancing prematurely to avoid potential temporal causality violations, as elaborated in §3. We further discuss this gap in §6.

A similar trend is observed in benchmarks conducted on A100 GPUs with larger models. AI Metropolis achieves a $2.45\times$ and $1.45\times$ speedup compared to *single-thread* and *parallel-sync*, respectively, and attains 82% of the *oracle* performance on 8 GPUs. Additional speedups are anticipated with higher data parallelism, given the *oracle-to-critical* ratio of 64.7% on A100s versus 88% on L4 GPUs, as memory demands for 70B models ($8.75\times$ higher) limit processing capacity.

4.3 Scaling up to a Thousand Agents

Given the limited research on accommodating hundreds of agents, we simulate a larger environment by concatenating multiple SmallVilles into a single, LargeVille for benchmarking. Agents in each segment replay different traces that we collected independently, but they operate within the same time and space. Since the concatenation approach introduces straightforward parallelism, rather than focusing on the critical path, which is artificially shortened due to the lack of interaction between different parts of the LargeVille, we introduce *no-dependency* as a more suitable lower bound for completion time when scaling agents. In this setting, all LLM calls can be issued simultaneously, maximizing hardware utilization. In Figure 5, 7 and 6, the *gpu-limit* uses the shorter completion time of the *critical* and *no-dependency* settings. Moreover, for benchmark with a larger number of agents, we opted to focus on two specific intervals from an entire day's simulation, as illustrated in Figure 4c: the busy hour (12 PM - 1 PM, approximately 5,000 calls) and the quiet hour (6 AM - 7 AM, approximately 800 LLM calls). This setup shortens experiment time and highlights scaling effects across different workloads, where busy hours feature long conversations, and quiet hours are mainly routine activities with less LLM queries as agents just wake up.

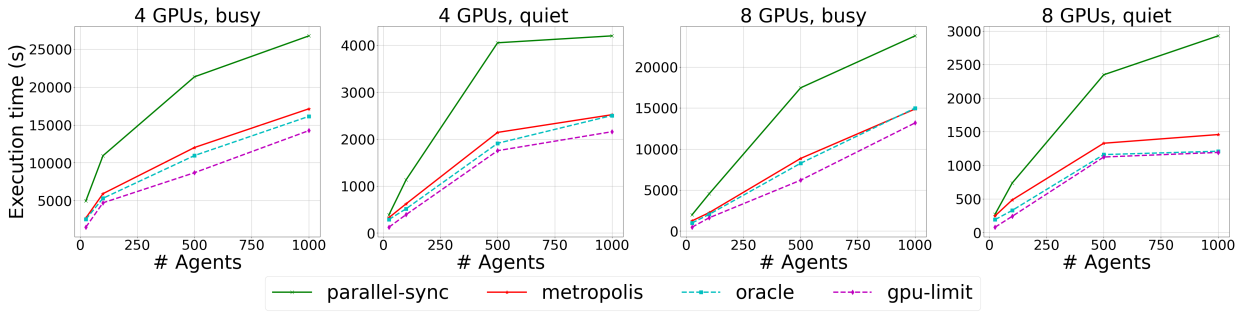


Figure 6. Benchmark of busy (12a.m. - 1p.m.) and quiet (6a.m. - 7a.m.) hour using Llama-3-70b-instruct on NVIDIA A100 GPUs with scaling number of agents from 25 to 1000.

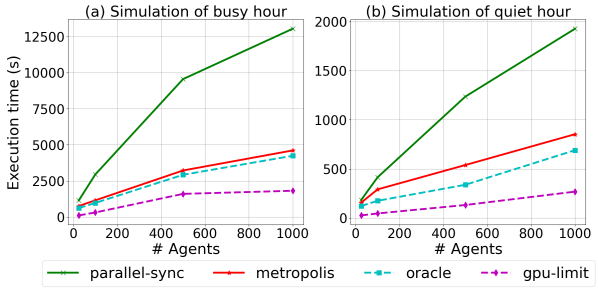


Figure 7. Benchmark of Mistral 8×7 on 8 A100 GPUs, with agent counts scaled from 25 to 1000.

The benefits of AI Metropolis increase with increasing numbers of agents. Figure 5 shows that AI Metropolis achieves closer performance to *oracle* as the number of agents increase: it achieves 90% of *oracle* on one GPU with 100 agents, reaching parity with *oracle* at 500 agents. On 8 GPUs, AI Metropolis improves from 53.1% to 97.0% of *oracle* across settings. Speedups over *single-thread* and *parallel-sync* also scale with agent count, increasing from 3.37× and 1.88× at 25 agents to 19.5× and 4.15× at 500 agents. Unlike *single-thread*, which cannot leverage parallelism, and *parallel-sync*, which suffers from costly synchronization, AI Metropolis utilizes parallelism more effectively, maximizing speedup as agent count grows.

After reaching peak speedup over *parallel-sync* at 500 agents, the speedup plateaus, slightly decreasing to 3.94× at 1000 agents. This is because, as agent count grows relative to available computational resources, even less efficient dependency management achieves adequate hardware utilization. Meanwhile, AI Metropolis reaches 97% of *oracle* performance, indicating that additional parallelism is less effective. This trend appears earlier on a single L4 GPU, where computational resources are more limited. AI Metropolis achieves a maximum speedup of 1.87× over *parallel-sync* at 100 agents, tapering to 1.60× as AI Metropolis's performance approaches *oracle*—from 90.9% at 100 agents to 100% at 500 agents.

Similar trends appear in the quiet hour benchmark, as shown in Figure 5, with some variation: the lighter and less frequent LLM calls in the quiet hour benchmark reduce the synchronization overhead for *parallel-sync*, allowing more parallelism. As a result, AI Metropolis shows a smaller speedup over *parallel-sync* with the same agents and GPUs—for instance, 1.28× in the 25-agent, 8-GPU setting, where achieved parallelism is 2.25 for *parallel-sync* and 2.80 for AI Metropolis. By comparison, the busy hour benchmark achieves parallelism values of 1.89 and 3.74 on the same setting, respectively. Nevertheless, as the number of agents increases, the speedup for AI Metropolis rises from 1.28× to 2.79× at 500 agents on 8 GPUs.

Similar trends also hold for larger models on 8 A100 GPUs. AI Metropolis peaks at a 1.97× speedup over *parallel-sync* with 500 agents in the busy hour benchmark and 2.01× in the 1000-agent quiet hour benchmark, as shown in Figure 6. To further explore model variability, we benchmarked the Mistral MoE $8 \times 7b$ model on the same 8 A100 platform, which uses 80% of a 70b model's memory with lighter I/O and computation. With the $8 \times 7b$ MoE model, we observe higher peak speedups of 2.97× and 2.29× over *parallel-sync* at 500 agents for busy and quiet hour benchmarks, respectively, due to greater resource availability on the GPUs, which allows for better parallelism utilization.

4.4 Priority Scheduling Breakdown

# GPUs	metropolis		oracle	
	4	8	4	8
w/ priority (s)	8611	6148	8392	5683
w/o priority (s)	8942	7114	8484	5689
Speedup (%)	3.84%	15.7%	1.10%	0.11%

Table 1. Performance breakdown of *metropolis* and *oracle* with and without priority scheduling on L4 GPUs. The first two rows are completion time in seconds.

All the experiments discussed so far have priority scheduling

enabled, where every request includes a step count, and requests with smaller counts have higher execution priority. This applies to the *oracle* baseline as well. We repeated the experiment of busy hours with 500 agents on 4 and 8 L4 GPUs for AI Metropolis and the *oracle*, but with priority scheduling turned off.

As shown in Table 1, priority scheduling does not significantly impact performance of *oracle* because it already achieves sufficient parallelism, and its dependency graph is sparse as discussed in §2.2, making priority less critical for unlocking additional parallelism. In contrast, we observed up to a 15.7% speedup for AI Metropolis with priority scheduling. This is because the conservative rules defined in §3.2 make agents falling behind to block others more frequently. Priority scheduling reduces this blocking, allowing AI Metropolis to perform closer to the *oracle*. With priority enabled, the average achieved parallelism in the 500-agent, 8-GPU benchmark increases from 41.9 to 50.9 for AI Metropolis, compared to a minor increase from 69.4 to 69.9 for *oracle*.

5 Related Work

LLM Agent Society Simulation and Multi-agents Collaboration. With increasing interest in LLM agents, several recent frameworks, such as Camel (Li et al., 2023), AutoGPT (Significant Gravitas, 2023), and OpenAI Swarm (OpenAI, 2024d), have emerged to simplify the development of multi-LLM agent interactions. However, these frameworks primarily provide interfaces for connecting LLM agents but lack a shared environment or state synchronization, making multi-agent interactions more akin to microservices connected via remote procedure calls. In contrast, GenAgent and similar explorations (Wang et al., 2023a; Gong et al., 2023; Altera.AL et al., 2024) represent a different approach, which we call *AI agent society*. In these systems, agents exist within a virtual world, interacting with both each other and the environment. However, this line of research typically emphasizes agent architectures and prompt engineering, often resulting in a lock-step simulation process for simplicity, which can lead to slower simulations. AI Town (a16z infra, 2023), though inspired by generative agents and similar to AI society frameworks, provides a platform with minimal environmental interactivity. Agents can move and converse with others, but cannot interact with the environment. Due to the unpredictable nature of LLM-powered conversations, there is no cohesive timeline: agents can talk for hours or just seconds. As a result, there is no structured progression of time, such as morning, afternoon, or night, where agents engage in different activities. Instead, agents simply walk and talk, making AI Town more of a conversation simulator than a dynamic AI society. By contrast, rather than simplifying the virtual world or compromising its functionality, AI Metropolis overcomes inherent performance challenges in the simulation of AI agent society through out-of-order execution, enhancing efficiency while preserving the logical correctness of the original synchronous simulation.

General Multi-agent Simulation. While LLM agent simulation may appear superficially similar to general multi-agent simulations, such as those used in reinforcement learning (Brockman et al., 2016; Zhu et al., 2024; Shacklett et al., 2023) and multi-agent processing (Emau et al., 2011), they present fundamentally different scheduling challenges due to the high per-agent computational demands and significant workload imbalances inherent in LLM execution, as discussed in §2.2. These unique demands necessitate specialized scheduling strategies. AI Metropolis is designed to offer a user experience comparable to that provided by established reinforcement learning frameworks, while seamlessly managing the additional complexities behind the scenes.

LLM Serving Optimizations. An active line of research (Yu et al., 2022; Kwon et al., 2023; Agrawal et al., 2023; Zheng et al., 2024; Sheng et al., 2023a; Chen et al., 2024) in LLM inference optimization has been making continuous advancements in increasing throughput and reducing latency from various angles. These engines typically achieve optimal performance when there are ample requests available for scheduling but generally do not address request dependencies. AI Metropolis complements these efficient engines by increasing parallelism within the simulation, thereby enhancing overall throughput. Since AI Metropolis decouples the simulation engine from the serving engine, improvements in serving engines directly boost simulation throughput. While a substantial body of research exists on general inference and serving, it remains largely orthogonal to our work. Some studies address dependencies among LLM requests, such as (Kim et al., 2023) for function calling and (Zheng et al., 2024) for LLM programs to achieve higher parallelization. However, these approaches assume rigid dependencies, unlike AI Metropolis, which focuses on reducing false dependencies to further enhance efficiency.

6 Discussion and Future Work

Applications of AI Metropolis. Although this paper highlights GenAgent as the primary use case, AI Metropolis offers broad applicability across various domains. First, as a pioneering framework for simulating social behaviors with LLMs, GenAgent has significantly influenced subsequent research, inspiring various studies with similar architectures as discussed in §5. By providing an efficient execution engine, AI Metropolis can benefit a broad audience and can be easily adapted to diverse simulation environments through parameter adjustments such as perception radius

and movement speed. Second, the core assumption behind AI Metropolis,that global synchronization is unnecessary because each agent perceives only a limited portion of the environment, is broadly applicable. This assumption mirrors real-world situations where humans or robots naturally have restricted perceptual fields. For example, video games commonly render only elements visible to players to optimize computational efficiency. Hence, AI Metropolis is suitable for a broad class of simulations aimed at replicating realistic, human-like interactions. Finally, while our current formulation emphasizes temporal-spatial relationships in Euclidean space, it can also be generalized to non-Euclidean spaces. For example, project OASIS (Yang et al., 2024) uses LLM agents to simulate interactions within social networks, where agent relationships are defined by network hops rather than physical distances. By mapping these hops to distances within AI Metropolis's framework, similar out-of-order execution mechanisms could be applied, potentially enhancing parallelism and improving simulation efficiency.

Offline and Interactive. While AI Metropolis currently focuses on maximizing throughput for offline simulation, its core principles—fine-grained dependency management and priority scheduling—are also applicable to interactive environments such as video games. The key distinction between a real game and an offline simulation lies in interactivity, which introduces strict latency requirements. We view games like The Sims (Electronic Arts, 2000) as hybrids of interactive and offline components: the player-facing elements demand low latency for real-time responsiveness, while background agents can operate in a simulation-driven manner to produce realistic social behaviors. A promising direction for AI Metropolis is to support such hybrid deployments, balancing request prioritization and incorporating lightweight decision-making components to reduce latency in interactive tasks while optimizing throughput for background simulations.

Conservative or Speculative Execution. As discussed in §3.2, AI Metropolis adopts conservative rules to prevent causality violations, which can extend the critical path and limit parallelism. Despite this, AI Metropolis delivers performance close to the *oracle* setting in most cases, as demonstrated in §4. This is largely due to its ability to expose sufficient parallelism and leverage effective priority scheduling. While there are scenarios where unlocking additional parallelism could further improve performance, the gap between AI Metropolis and the *oracle* remains small, as evidenced by the quiet hour benchmark in §4.3. Nonetheless, this gap highlights opportunities for further optimization. Incorporating speculative execution with race detection could help close it, although this might challenge the system's scalability principles—an interesting trade-off we reserve for future research.

Simulation with Smaller Models. While models with fewer parameters or quantized parameters can achieve higher throughput, they typically require even greater parallelism to fully utilize hardware resources. Additionally, our experiments show that smaller models often struggle to accurately follow instructions in complex tasks. Although orthogonal to the current design of AI Metropolis, addressing this trade-off between accuracy and throughput represents a promising direction for future work.

Online APIs and Local Models. While proprietary APIs like GPT-4 (OpenAI, 2024c) and Claude 3 (Anthropic, 2024) lead in performance, open-source models are on the rise. AI Metropolis supports local model serving with optimized dependency management for faster, cost-effective simulations, yet remains compatible with online APIs, enhancing parallelism and simplifying state management for users.

7 Acknowledgment

We thank anonymous reviewers and our shepherd, Prof. Dimitrios Stamoulis, for their constructive suggestions. We thank Joon Sung Park, Brennan Shacklett, Mark Zhao, Athinagoras Skiadopoulos, Qizheng Zhang, Piero Molino and the (Altera.AL) team for valuable discussions and feedback. This research was partly supported by the Stanford Platform Lab and its affiliates, and by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. This research was also partly supported by NSF CNS-2047283.

References

a16z infra. Ai town: An open-source platform inspired by generative agents. https://github.com/a16z-infra/ai-town, 2023.

Agrawal, A., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B. S., and Ramjee, R. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.

Altera.AL. Building digital humans. https://altera.al. Accessed: 2024-11-04.

Altera.AL, Ahn, A., Becker, N., Carroll, S., Christie, N., Cortes, M., Demirci, A., Du, M., Li, F., Luo, S., Wang, P. Y., Willows, M., Yang, F., and Yang, G. R. Project sid: Many-agent simulations toward ai civilization, 2024. URL https://arxiv.org/abs/2411.00114.

Anthropic. Claude 3: An overview. https://www.anthropic.com/claude-3, 2024.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J.,

Schulman, J., Tang, J., and Zaremba, W. Openai gym, 2016.

Chen, L., Ye, Z., Wu, Y., Zhuo, D., Ceze, L., and Krishnamurthy, A. Punica: Multi-tenant lora serving. *Proceedings of Machine Learning and Systems*, 6:1–13, 2024.

Chiang, W.-L., Zheng, L., Sheng, Y., Angelopoulos, A. N., Li, T., Li, D., Zhang, H., Zhu, B., Jordan, M., Gonzalez, J. E., and Stoica, I. Chatbot arena: An open platform for evaluating llms by human preference, 2024. URL https://arxiv.org/abs/2403.04132.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems. *CoRR*, abs/2110.14168, 2021. URL https://arxiv.org/abs/2110.14168.

Dubois, Y., Li, C. X., Taori, R., Zhang, T., Gulrajani, I., Ba, J., Guestrin, C., Liang, P. S., and Hashimoto, T. B. Alpacafarm: A simulation framework for methods that learn from human feedback. *Advances in Neural Information Processing Systems*, 36, 2024.

Electronic Arts. The sims, 2000. URL https://www.ea.com/games/the-sims. Video game.

Emau, J., Chuang, T., and Fukuda, M. A multi-process library for multi-agent and spatial simulation. In *Proceedings of 2011 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pp. 369–375. IEEE, 2011.

Gong, R., Huang, Q., Ma, X., Vo, H., Durante, Z., Noda, Y., Zheng, Z., Zhu, S.-C., Terzopoulos, D., Fei-Fei, L., and Gao, J. Mindagent: Emergent gaming interaction. 2023. URL https://arxiv.org/abs/2309.09971.

Grossmann, I., Feinberg, M., Parker, D. C., Christakis, N. A., Tetlock, P. E., and Cunningham, W. A. Ai and the transformation of social science research. *Science*, 380(6650): 1108–1109, 2023.

Guadarrama, S., Korattikara, A., Ramirez, O., Castro, P., Holly, E., Fishman, S., Wang, K., Gonina, E., Wu, N., Kokiopoulou, E., Sbaiz, L., Smith, J., Bartók, G., Berent, J., Harris, C., Vanhoucke, V., and Brevdo, E. TF-Agents: A library for reinforcement learning in tensorflow. https://github.com/tensorflow/agents, 2018. URL https://github.com/tensorflow/agents. [Online; accessed 25-June-2019].

Kim, S., Moon, S., Tabrizi, R., Lee, N., Mahoney, M. W., Keutzer, K., and Gholami, A. An llm compiler for parallel function calling. *arXiv preprint arXiv:2312.04511*, 2023.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Li, G., Hammoud, H. A. A. K., Itani, H., Khizbullin, D., and Ghanem, B. Camel: Communicative agents for "mind" exploration of large language model society. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

Liu, R., Yang, R., Jia, C., Zhang, G., Yang, D., and Vosoughi, S. Training socially aligned language models on simulated social interactions. In *The Twelfth International Conference on Learning Representations*, 2023.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models, 2016.

Meta. Llama 3: Advancing ai research. https://llama.meta.com/llama3/, 2024. Accessed: 2024-06-25.

Mistral AI. https://mistral.ai/news/mixtral-of-experts/, 2023.

Narayan, S., Cohen, S. B., and Lapata, M. Don't give me the details, just the summary! topic-aware convolutional neural networks for extreme summarization, 2018.

OpenAI. Chatgpt: Openai language model. https://chat.openai.com/, 2024a. Accessed: 2024-06-22.

OpenAI. Openai gpt-3.5 api. https://platform.openai.com/docs/models/gpt-3-5, 2024b. Accessed: 2024-06-25.

OpenAI. Introducing gpt-4. https://openai.com/research/gpt-4, 2024c.

OpenAI. Swarm: Scalable infrastructure for multi-agent coordination, 2024d. URL https://github.com/openai/swarm. Accessed: 2024-10-29.

Park, J. S. Generative agents. https://github.com/joonspk-research/generative_agents, 2024. GitHub repository.

Park, J. S., O'Brien, J. C., Cai, C. J., Morris, M. R., Liang, P., and Bernstein, M. S. Generative agents: Interactive simulacra of human behavior. In *In the 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23)*, UIST '23, New York, NY, USA, 2023. Association for Computing Machinery.

Redis. Redis: In-memory data structure store. https://redis.io. Accessed: 2024-06-25.

Shacklett, B., Rosenzweig, L. G., Xie, Z., Sarkar, B., Szot, A., Wijmans, E., Koltun, V., Batra, D., and Fatahalian, K. An extensible, data-oriented architecture for high-performance, many-world simulation. *ACM Transactions on Graphics (TOG)*, 42(4):1–13, 2023.

Sheng, Y., Cao, S., Li, D., Hooper, C., Lee, N., Yang, S., Chou, C., Zhu, B., Zheng, L., Keutzer, K., Gonzalez, J. E., and Stoica, I. S-lora: Serving thousands of concurrent lora adapters. *arXiv preprint arXiv:2311.03285*, 2023a.

Sheng, Y., Cao, S., Li, D., Zhu, B., Li, Z., Zhuo, D., Gonzalez, J. E., and Stoica, I. Fairness in serving large language models. *arXiv preprint arXiv:2401.00588*, 2023b.

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

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. *arXiv preprint arXiv:2305.16291*, 2023a.

Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Chen, X., Lin, Y., Zhao, W. X., Wei, Z., and Wen, J. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6), March 2024. ISSN 2095-2236. doi: 10.1007/s11704-024-40231-1. URL http://dx.doi.org/10.1007/s11704-024-40231-1.

Wang, Z., Cai, S., Chen, G., Liu, A., Ma, X., and Liang, Y. Describe, explain, plan and select: Interactive planning with large language models enables open-world multitask agents. *arXiv preprint arXiv:2302.01560*, 2023b.

Xi, Z., Chen, W., Guo, X., He, W., Ding, Y., Hong, B., Zhang, M., Wang, J., Jin, S., Zhou, E., Zheng, R., Fan, X., Wang, X., Xiong, L., Zhou, Y., Wang, W., Jiang, C., Zou, Y., Liu, X., Yin, Z., Dou, S., Weng, R., Cheng, W., Zhang, Q., Qin, W., Zheng, Y., Qiu, X., Huang, X., and Gui, T. The rise and potential of large language model based agents: A survey, 2023. URL https://arxiv.org/abs/2309.07864.

Yang, Z., Zhang, Z., Zheng, Z., Jiang, Y., Gan, Z., Wang, Z., Ling, Z., Chen, J., Ma, M., Dong, B., et al. Oasis: Open agents social interaction simulations on one million agents. *arXiv preprint arXiv:2411.11581*, 2024.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, 2022.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C., and Sheng, Y. Sglang: Efficient execution of structured language model programs, 2024. URL https://arxiv.org/abs/2312.07104.

Zhu, Z., de Salvo Braz, R., Bhandari, J., Jiang, D., Wan, Y., Efroni, Y., Wang, L., Xu, R., Guo, H., Nikulkov, A., Korenkevych, D., Dogan, U., Cheng, F., Wu, Z., and Xu, W. Pearl: A production-ready reinforcement learning agent, 2024. URL https://arxiv.org/abs/2312.03814.

Ziems, C., Held, W., Shaikh, O., Chen, J., Zhang, Z., and Yang, D. Can large language models transform computational social science? arxiv. Technical report, Retrieved 2023-10-06, from http://arxiv. org/abs/2305.03514, 2023.