MATRIX: PEER-TO-PEER MULTI-AGENT SYNTHETIC DATA GENERATION FRAMEWORK

Dong Wang [1] **Yang Li** [1] **Ansong Ni** [1] **Ching-Feng Yeh** [1] **Youssef Emad** [1] **Xinjie Lei** [1] **Liam Robbins** [1] **Karthik Padthe** [1] **Hu Xu** [1] **Xian Li** [1] **Asli Celikyilmaz** [1] **Ramya Raghavendra** [1] **Lifei Huang** [1] **Carole-Jean Wu** [1] **Shang-Wen Li** [1]

ABSTRACT

Synthetic data has become increasingly important for training large language models, especially when real data is scarce, expensive, or privacy-sensitive. Many such generation tasks require coordinated multi-agent workflows, where specialized agents collaborate to produce data that is higher quality, more diverse, and structurally richer. However, existing frameworks for multi-agent synthesis often depend on a centralized orchestrator, creating scalability bottlenecks, or are hardcoded for specific domains, limiting flexibility. We present **Matrix**, a decentralized framework that represents both control and data flow as serialized messages passed through distributed queues. This peer-to-peer design eliminates the central orchestrator. Each task progresses independently through lightweight agents, while compute-intensive operations, such as LLM inference or containerized environments, are handled by distributed services. Built on Ray, Matrix scales to tens of thousands of concurrent agentic workflows and provides a modular, configurable design that enables easy adaptation to a wide range of data generation workflows. We evaluate Matrix across diverse synthesis scenarios, such as multi-agent collaborative dialogue, web-based reasoning data extraction, and tool-use trajectory generation in customer service environments. In all cases, Matrix achieves 2–15× higher data generation throughput under identical hardware resources, without compromising output quality.

1 INTRODUCTION

Large scale machine learning models, such as large language models (LLMs) and multi-modal foundation models, are increasingly trained with synthetic data to reduce dependence on costly, noisy, or privacy-sensitive human-curated datasets (Grattafiori et al., 2024; Abdin et al., 2024; Betker et al., 2023). Recent advances have shifted toward agentic synthetic data generation, where data is produced through interactions among multiple intelligent agents rather than a single model or fixed pipeline. This paradigm enables multi-agent collaboration for diverse generation tasks such as code synthesis, instruction and dialogue creation, knowledge-grounded question answering, and multi-modal content generation. In these settings, the workflows often involve complex control flows with loops, moving beyond traditional linear data generation pipelines. For example, Kimi K2 (Bai et al., 2025) employs a large-scale multi-agent data synthesis pipeline to construct diverse tool-use and reasoning demonstrations. Similarly, CWM (Copet et al., 2025) leverages autonomous software engineering agents to generate multi-step trajectories for code understanding and debugging. These systems exemplify the growing adoption of multi-agent pipelines for synthetic data generation in large scale LLM training, underscoring the need for flexible and scalable frameworks for data synthesis.

Generic agent frameworks such as AutoGen (Wu et al., 2023; Fourney et al., 2024), LangGraph (LangChain, 2025), and CrewAI (CrewAI, 2025) provide convenient abstractions for *authoring* agent workflows and expressing control flow. However, operating these workflows at the throughput regime targeted in this work typically requires additional production scaffolding beyond the workflow definition itself, including scalable LLM/tool services, backpressure and concurrency control, retries/timeouts, and logging/metrics. Accordingly, rather than presenting an end-to-end throughput bake-off against these authoring-focused frameworks, we view them as complementary: Matrix can reuse their workflow specifications (e.g., LangGraph-style state graphs) by packaging common patterns as reusable orchestrator subclasses within our runtime, enabling users to design control flow in familiar form while benefiting from Matrix's decentralized scheduling and distributed service integration.

Recently a number of systems have been developed to gen-

[1]Meta FAIR. Correspondence to: Dong Wang <dwoanngg@gmail.com>, Shang-Wei Li <shangwel@meta.com>.

Proceedings of the 9th MLSys Conference, Bellevue, WA, USA, 2026. Copyright 2026 by the author(s).

erate synthetic data for *particular* agentic settings. Notable examples include AgentInstruct (Mitra et al., 2024), SWE-Agent (Yang et al., 2024), SWE-Synth (Pham et al., 2025), TaskCraft (Shi et al., 2025), and AgentSynth (Xie et al., 2025). These systems demonstrate that carefully designed agent roles and validation loops can produce high-quality data and strong task-level performance. However, their implementations are often optimized around a *specific* task structure (e.g., software engineering, tool execution, or instruction generation), so adapting them to new domains can require non-trivial engineering, such as refactoring the control-flow logic, state representation, and service integration, rather than simply swapping prompts or models. At scale, running many independent workflow instances also introduces systems challenges. Users typically rely on per-process concurrency (threads/async) and/or external job orchestration (e.g., Kubernetes Jobs, Airflow, or distributed task queues) to execute large numbers of tasks concurrently, and must additionally provision shared services for inference, tool execution and logging.

To address these limitations, we present Matrix, a distributed runtime for scalable, multi-agent synthetic data generation and agentic experimentation. Matrix frames data generation as a *data-to-data transformation*: each input row represents an independent task, and the runtime executes many such tasks concurrently, each running its own agentic workflow.

The core idea behind Matrix is a peer-to-peer (P2P) agent architecture that replaces centralized orchestration with decentralized, message-driven scheduling. The state of each task, which includes orchestration logic, intermediate results, and conversation history, is serialized into messages that are passed among agents. The active agent consumes and updates this state, then emits it to the next agent determined by the orchestrator. Because agents themselves are stateless, they can scale elastically and independently across the cluster.

Unlike traditional batch-level scheduling in distributed execution engines such as Spark (Zaharia et al., 2012) and Ray Data (Moritz et al., 2018), where the pipeline controls progress across synchronized batches, Matrix performs row-level scheduling through peer-to-peer message orchestration. Control and data flow are embedded in messages, allowing each task to progress asynchronously through agents. This eliminates idle periods caused by batch-level barriers.

Distributing a centralized workflow with an external job scheduler (e.g., SLURM or Kubernetes) can sidestep a single orchestrator bottleneck by running many workflow replicas in parallel. In practice, however, scaling this way typically pushes substantial distributed-systems work into each application: partitioning the input into jobs, provisioning and load-balancing shared services (LLM inference, tool execution, logging, and state), implementing retries and timeouts, and tuning coupled parameters such as number of jobs, per-job concurrency. Matrix instead provides a self-contained runtime for agentic data generation: workflow state and control are carried as messages, scheduling is decentralized at row granularity, and heavy computation is delegated to shared distributed services with built-in backpressure. This reduces application-specific orchestration glue while scaling to tens of thousands of concurrent workflows.

Key Contributions.

1. We introduce **Matrix**, a **scalable runtime** for large scale multi-agent synthetic data generation capable of efficiently executing tens of thousands of concurrent workflows. Matrix adopts a **peer-to-peer agent architecture** with message-embedded control and state representation, eliminating centralized orchestration bottlenecks and idle time caused by batch-level synchronization. This design enables fully asynchronous and fine-grained execution at scale.

2. Matrix is designed to be **flexible** and **extensible**, supporting diverse multi-agent use cases. Its modular architecture separates key components, including the generation loop and distributed services for LLM inference and containerized execution, and the entire system is fully configurable through Hydra.

3. We evaluate Matrix on three representative case studies: Collaborative Reasoner (Ni et al., 2025a), NaturalReasoning (Yuan et al., 2025), and Tau2-bench (Barres et al., 2025a). Matrix achieves **2–15**$\times$ **higher token throughput** than specialized baseline systems while maintaining comparable output quality.

4. Matrix is built entirely on an **open source stack**, including SLURM, Ray, vLLM, SGLang, and Apptainer. It supports both open-weight models and LLM API proxies. We have open sourced the framework to the community to foster open development and collaborative research.

2 Related Work

LLM and agentic benchmarks. LLMs are commonly evaluated on reasoning benchmarks such as Math-500 (Hendrycks et al., 2021) and MMLU-Pro (Wang et al., 2024). Recent multi-agent systems run on standardized benchmarks that test complex, multi-step reasoning and tool use. Examples include SWE-bench (Jimenez et al., 2024), Tau2-Bench (Barres et al., 2025a), MCP-Bench (Wang et al., 2025), and MLE-Bench (Chan et al., 2025). Each benchmark comes with a reference agentic system that solves the tasks, such as SWE-agent and Tau2-agent. In this work, we

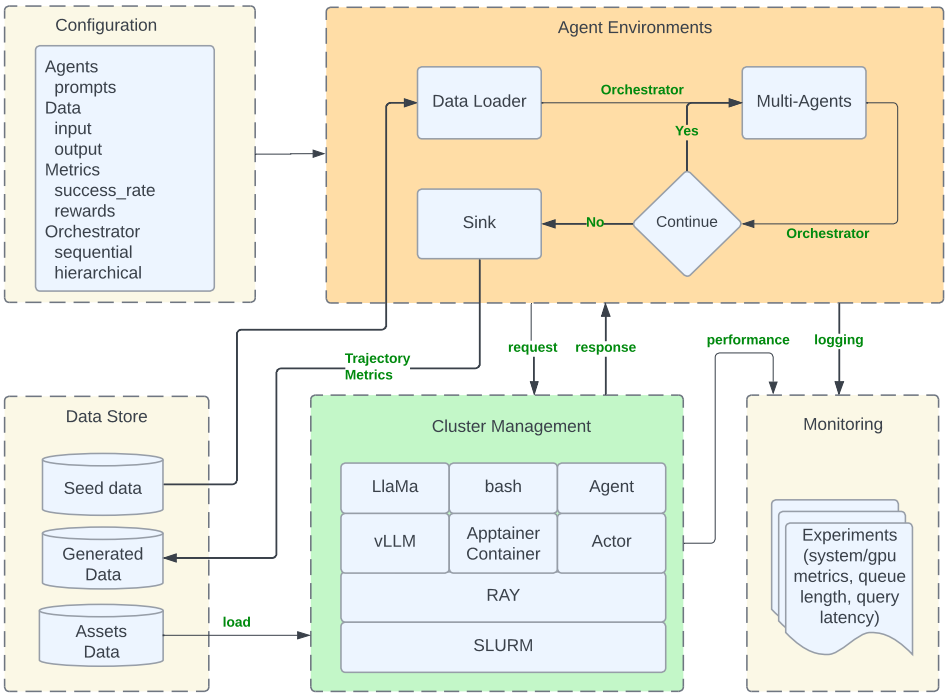

Figure 1. Matrix Agentic Data Generation Architecture.

use Tau2-Bench and MMLU-Pro as sources of initial tasks to generate agent trajectories that can be used for fine-tuning LLMs.

Data Synthesis via Multi-agents Workflows. The scarcity of high-quality agentic training data has led to the development of synthetic data generation techniques employing multi-agent frameworks. AgentInstruct (Mitra et al., 2024) generates multi-turn instruction-response data by coordinating multiple agents to propose, verify, and refine synthetic tasks based on seed examples. TaskCraft (Shi et al., 2025) automatically generates multi-step, multi-tool agentic tasks with verifiable execution trajectories.APIGen-MT (Prabhakar et al., 2025) is a two-phase framework that generates verifiable, multi-turn agent interaction data. While these frameworks are tailored to specific data needs and emphasize data quality, our approach offers a generic framework capable of supporting multiple use cases with a focus on scalability.

Peer-to-Peer ML Systems Peer-to-peer (P2P) architectures have long been foundational in distributed computing and communications. In ML, P2P systems have been leveraged to enhance scalability, privacy, and personalization. For instance, The SPIRT (Barrak et al., 2023) framework introduces a serverless P2P ML training architecture that leverages RedisAI for in-database operations, achieving significant reductions in model update times and demonstrating resilience against peer failures. Similarly, BlockDFL (Qin et al., 2024) employs blockchain-based coordination to facilitate fully decentralized federated learning, incorporating mechanisms to defend against poisoning attacks and reduce communication costs. While prior P2P ML systems focus on efficient training and privacy-preserving computation, Matrix introduces a general framework using P2P communication to coordinate agent workflows for scalable multi-agent data synthesis.

3 Matrix Overview

This section provides an overview of the Matrix framework, describing its system architecture and the core algorithm that enables scalable, asynchronous multi-agent synthetic data generation.

3.1 System Architecture

Figure 1 illustrates the architecture of Matrix, a distributed runtime for multi-agent synthetic data generation. The system is designed to be **modular** and **configurable**, separating lightweight peer-to-peer agents from scalable backend services (e.g., LLM inference and containerized tool execution) so that components can be adapted and scaled independently across different workflows.

Cluster Management. The framework is deployed atop SLURM (Yoo et al., 2003), a widely adopted distributed computing environment, with a Ray (Moritz et al., 2018) cluster serving as the execution substrate. Ray Serve provides high-throughput LLM inference services, backed by vLLM (Kwon et al., 2023), SGLang (Zheng et al., 2024), and FastGen (FAIR, 2025). Containerized execution is supported through Apptainer (Kurtzer et al., 2017), enabling stateful environments to be launched on demand. Each agent is implemented as a Ray Actor, allowing scalable parallelization and fine-grained resource placement across worker nodes.

Configuration. System configurability is managed through Hydra (Yadan, 2019), which specifies agent roles, input–output schemas, generation metrics, and resource requirements (e.g., LLM engine selection). The configuration also defines the orchestrator responsible for control and data flow management. Users can define the use case specific orchestrator class in Hydra, then hand-write control flow as imperative if/else logic in the implemetation. Matrix also supports a graph-based specification. For users familiar with LangGraph (LangChain, 2025), we provide a `LangGraphOrchestrator` that performs state transitions based on a user-supplied LangGraph: users define nodes (agent roles) and edges (transition rules) in the standard LangGraph API, and the orchestrator updates its state by executing the graph's routing decision at each step. This makes non-sequential workflows easier to express and to visualize while preserving Matrix's decentralized, message-driven execution model.

Agents Environments. In the peer-to-peer generation process, each input datum is encapsulated into an orchestrator instance and passed to the initial agent. The agent processes the instance, updates the orchestrator state, and forwards control to the next designated agent. This process continues iteratively until completion. A detailed algorithmic description is provided in Section 3.2. We will use Matrix to build different agents environments in the experiments section.

Monitoring. Logging and observability are critical for debugging and performance analysis. Matrix integrates with Grafana (Labs, 2025) for real-time monitoring. In addition to standard performance metrics, it provides custom indicators such as distributed queue length and the number of pending asynchronous tasks. These metrics help identify throughput bottlenecks and evaluate overall system health.

3.2 Data Generation Algorithm

Algorithm 1 illustrates the core workflow of Matrix's peer-to-peer agentic generation runtime. The system begins by reading the Hydra configuration `cfg`, which specifies all agent roles and their resource requirements (e.g., CPU, GPU, and memory). As shown in Lines 19–24, the function `create_team()` instantiates a distributed team of Ray actors for each agent role, allowing heterogeneous resource allocation across agent types.

Algorithm 1 Matrix P2P agentic generation pseudocode.

```
1  @ray.remote
2  class AgentActor:
3    async def _event_loop(self, team):
4      while True:
5        orch = await self.queue.get()
6        result = self.process(orch)
7        orch.update(result)
8        next = orch.current_agent()
9        random.choice(team[next]).send(orch)
10
11 class SequentialOrchestrator:
12   def update(self, result):
13     self.history.append(result)
14     self.idx = (self.idx + 1) % len(self.order)
15
16   def current_agent(self):
17     return "_sink" if self.is_done else self.order[self.
         idx]
18
19 def create_team(cfg):
20   return {
21     role: [ray_create_actor(role, c)
22           for _ in range(c.num_instances)]
23     for role, c in cfg.items()
24   }
25
26 team = create_team(cfg.agents)
27 for item in dataset:
28   orch = Orchestrator(item)
29   first = random.choice(team[orch.current_agent])
30   first.send(orch)
```

Agent EventLoop. The main generation loop (Lines 26–30) iterates over input items in the dataset. For each item, an `Orchestrator` object is created to manage task-specific state and control flow. The orchestrator is initially dispatched to the first agent in the sequence, sampled randomly from the corresponding role group. Each agent runs as a persistent event-driven process (Lines 3–9) implemented by the `AgentActor` class. Within its asynchronous `_event_loop`, the agent dequeues orchestrators from its inbox, applies role-specific logic through `process()`, updates task state, and forwards it to the next designated agent.

Orchestration. An example orchestrator `SequentialOrchestrator` is in Lines 11–17. It maintains a structured `history` of intermediate results and a configurable `order`, which determines the sequence of participating agents. After each interaction, `update()` advances the internal index to the next agent. The process continues cyclically until the orchestrator's `is_done` flag is set, at which point it is routed to a special terminal agent, `_sink`, for result persistence and metric aggregation.

Concurrency Control. Advanced runtime features (not shown for brevity) include task-level concurrency control

through a `max_concurrency` parameter and semaphore-based scheduling. The semaphore is decremented when an orchestrator is dispatched to the `first` agent and incremented upon completion by the `_sink` agent. This mechanism limits the number of active orchestrators, ensuring controlled resource utilization and stability during large-scale distributed execution. Note we rely on Ray to avoid race conditions in distributed RPC calls.

4 Agent Environment Design for Matrix

This section describes the system's internal design, including its orchestration model, distributed service layer, parallelism strategies, scheduling policies, fault tolerance mechanisms and network bandwidth optimization.

4.1 P2P Orchestration

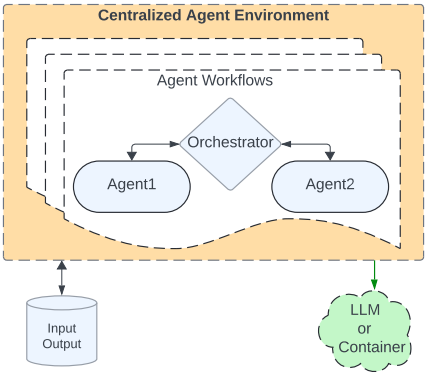

Figure 2. Traditional centralized orchestration.

As illustrated in Figure 2, centralized orchestration must manage execution order (control flow), message passing (data flow), and the full lifecycle of requests and responses for LLMs and containerized environments. Handling all of this for tens of thousands of concurrent workflows quickly becomes a scalability bottleneck. Matrix addresses this by representing workflows as serializable *orchestrators* that can be updated and exchanged among distributed agents (Figure 3). The *driver*, which runs the generation framework, plays a lightweight role: it simply publishes an orchestrator to start a task, enabling an asynchronous initiation model. Agents equipped with LLMs and tools consume messages, perform local actions, update both control and data states, and forward the updated orchestrator to the next agent. Execution continues until the orchestrator signals completion, at which point a designated sink collects the final message and persists it to the output dataset. Using P2P orchestration, Matrix avoids bottlenecks, improves scalability, and enables fully asynchronous execution among agents.

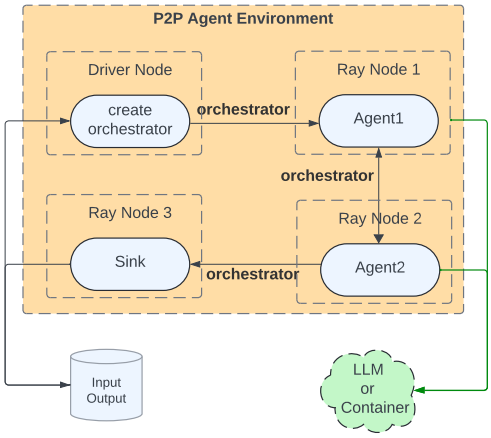

Figure 3. P2P Orchestration in Matrix.

4.2 Distributed Services

Matrix offloads computationally intensive tasks to distributed services, allowing them to scale independently of the agents. For LLM inference, Matrix employs gRPC-based communication to avoid HTTP overhead. Because the Ray head node can become network-bound, Matrix maintains a local cache of active model replica URLs, enabling direct load-balanced traffic through worker nodes. Sticky routing can reuse prefix cache for multi-turn long conversations. In addition to Huggingface models, proxies are built for commercial LLM API services. For stateful services such as Apptainer containers, agents acquire containers by ID to be able to route multiple commands to the same container instance, rather than a randomly selected one. This is managed via a *resource pool* and a *registry* that maps container IDs to Ray actors running the corresponding containers. This design allows agents to efficiently route messages and reuse shared resources.

4.3 Parallel Execution Strategies

Matrix supports multiple forms of parallelism to maximize scalability and cluster utilization.

- **Data parallelism.** Similar to distributed processing systems such as Spark (Zaharia et al., 2012) and Ray Data (Moritz et al., 2018), Matrix can partition large input datasets consisting of many small files for independent processing. For multi-file inputs, Matrix automatically distributes files across partitions. Datasets containing a few large files can be preprocessed into smaller shards to enable higher parallelism.

- **Task parallelism.** Multiple generation tasks can execute concurrently using asynchronous programming, threads, or processes. Matrix adopts an `asyncio`-

based model: the driver initializes orchestrators, and agents process tasks asynchronously. Since computationally heavy operations are offloaded to distributed services, lightweight agents can handle tens of thousands of concurrent tasks efficiently without I/O blocking.

- **Agent parallelism.** Each agent role is implemented as Ray actors with configurable CPU, GPU, and memory allocations. Roles can scale horizontally by launching multiple distributed agent instances, each processing assigned tasks independently. Ray system distributes these actors across cluster nodes, enabling each role to scale without the resource contention commonly seen in centralized orchestration.

For LLM-based agents, computational cost dominates over input pipeline overhead. Usually data loading is not a bottleneck (one exception is the NaturalReasoning task in Section 5.2). Matrix's peer-to-peer architecture and distributed services ensure efficient utilization of cluster resources even with moderate data and agent-level parallelism. This efficiency arises from Matrix's ability to run tens of thousands of asynchronous tasks concurrently, each processing one data item independently.

4.4 Row-Level Scheduling

In batch processing systems, such as Ray Data, tasks are grouped into fixed-size batches and executed by actors. While this approach can reduce per-task scheduling overhead for homogeneous workloads, it introduces inefficiencies when tasks have variable computational demands or diverging control flows. A long-running or complex task within a batch can keep the current batch running and stall the execution of subsequent batches, creating idle resources and underutilized GPUs. We refer to this phenomenon as *batch-level scheduling*.

In contrast, Matrix schedules each task independently as soon as prior tasks complete, a mechanism called *row-level scheduling*. Each orchestrator message representing a single task flows through the P2P agent network. This design eliminates the bubble effects inherent in batch processing, achieves higher GPU utilization, and reduces end-to-end latency for heterogeneous, multi-agent workloads. Row-level pipelining, combined with distributed services and asynchronous agent execution, is a key factor in Matrix's scalability and efficiency for large-scale data synthesis tasks.

4.5 Agent Fault Tolerance

Matrix currently provides *at-most-once* execution semantics. Tasks may fail for various reasons, including network errors, timeouts, and actor crashes. Failed tasks can be collected from the output dataset and re-run offline if needed. Matrix workflows are implemented by extending a base agent class, and use-case-specific logic may introduce bugs that crash an agent. Ray can restart crashed agent actors, however, any in-flight orchestrator messages that were dequeued by the crashed agent are not recoverable under at-most-once semantics. To track in-flight orchestrators and surface failures reliably, Matrix uses per-role message brokers. All agents of the same role share a broker, and all incoming and outgoing orchestrator messages for that role are routed through it. Each broker maintains (i) an incoming queue of orchestrators waiting to be processed and (ii) an assignment map that records which orchestrators are currently assigned to which agent instance. The broker dispatches orchestrators to agents in a round-robin manner. After an agent finishes processing an orchestrator, it returns the updated orchestrator to the broker, the broker then removes the corresponding entry from the assignment map and forwards the orchestrator to the next role's broker. When an agent crashes and is restarted by Ray, it re-registers with its broker. The broker detects that the previous instance has died, marks all orchestrators assigned to that instance as failed based on the assignment map, and forwards them to the sink for persistence as failed trajectories. With this design, use-case-specific agents can crash and restart without halting the system, as long as the brokers and sink remain available. Brokers and the sink are framework components (not customized per use case), and we rely on them for reliability. If a broker or the sink fails, the generation job terminates. To mitigate transient network issues, Matrix uses retries for communication between agents, brokers and sink.

4.6 Message Offloading

The orchestrator is serialized and exchanged among agents. As shown in Algorithm 1, its `history` field stores inter-agent conversations, which can be large. A common optimization is to offload this history to an external cache such as Redis. While this reduces orchestrator size, it simply shifts network traffic from occurring between agents to occurring between agents and the cache. Since the history is frequently updated and used for constructing LLM prompts, the total network bandwidth can actually double because each agent must retrieve, update, and store the complete history every turn.

Matrix instead retains the history structure within the orchestrator, while storing large conversation content that exceed a configurable size threshold in Ray's distributed object store. The history holds only the object identifiers, and content is retrieved on demand. Objects are immutable once stored, and all history-related objects are deleted when the orchestrator signals completion. This design keeps the orchestrator compact, reduces redundant transfers, and minimizes network load. Section 5.3.2 quantifies these benefits experimentally.

4.7 System Debugging

Debugging distributed systems is challenging, especially under peer-to-peer message passing. Matrix relies on structured logging and trajectory recording for debuggability. Ray streams actor logs back to the driver process, enabling a "local-like" debugging experience even when agents are distributed across the cluster. Matrix also records a full trajectory for each input task. When a task encounters an issue, the trajectory includes the relevant error context for offline analysis (e.g., timeouts, connection failures, and service errors). For unexpected exceptions, including agent implementation bugs, each agent runs an `asyncio` event loop and tracks pending futures. Unhandled exceptions propagate to the corresponding future. Matrix then marks the orchestrator as failed and routes it to the sink, which persists the failed trajectory to the output dataset. Users can subsequently filter failed trajectories and re-run them if needed.

5 Experiments

We evaluate Matrix across three case studies on synthetic data generation. Together, these experiments demonstrate the framework's scalability, robustness, and adaptability to diverse workloads. In this section, the terms "Matrix" and "P2P-agent" are used interchangeably to refer to the same framework.

5.1 Collaborative Reasoner (Coral)

Collaborative Reasoner (Coral) (Ni et al., 2025a) evaluates and improves multi-agent collaborative reasoning in LLMs through dialogue-driven tasks. Unlike single-agent evaluations, Coral requires two agents to discuss, disagree, and reach consensus over multi-turn interactions. Scalable training data is generated via self-collaboration, where an LLM plays both roles. In this work, we adopt the same agent setup, implemented as distributed agents in Figure 4.

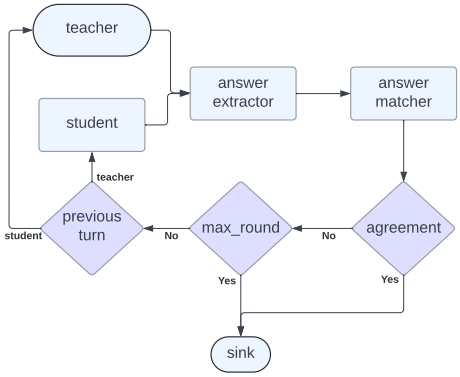

Figure 4. P2P-agents for Collaborative Reasoner.

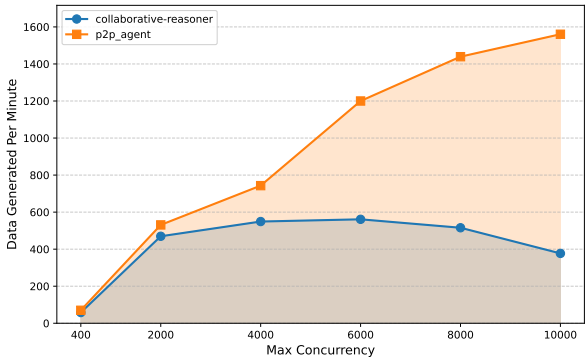

Figure 5. Scalability of P2P-agents vs Coral baseline.

We directly compare Matrix to the official Collaborative Reasoner implementation (Ni et al., 2025b) as the baseline. Both systems use asyncio for concurrency. The baseline framework uses a single orchestrator to coordinate thousands of concurrent generation tasks, while Matrix distributes coordination responsibilities across agents in a peer-to-peer fashion. To compare the two, we run the same number of MMLU-Pro questions by changing the number of A100 nodes, and in both cases use Llama-3.1-8B-Instruct (Grattafiori et al., 2024) as the underlying language model for all agents. Task concurrency is adjusted according to the number of A100 nodes as $50 \times N_{GPU}$, leveraging all 8 GPUs per node with 50 concurrent queries per GPU. As shown in Figure 5, the Matrix implementation scales almost linearly as more GPU nodes are added, while the centralized orchestration approach of the baseline system becomes a bottleneck and plateaus due to the overhead of scheduling a large number of asynchronous tasks from a single control point.

Table 1. P2P-Agent achieves **6.8**× higher token throughput than Coral baseline.

Metric	Baseline	P2P-Agent
Runtime	9:03:22	4:17:05
Concurrent tasks	5,000	12,400
Total trajectories	300k	1 Million
Agreement correctness	0.4732	0.4778
Tokens generated	616,759,036	2,002,025,810
Tokens per second	18,917	129,833

Large-Scale Results. We further tested both systems on 31 A100 nodes (248 GPUs) using LLaMA-3.1-8B-Instruct. For P2P-agent, we set the concurrency to $248 \times 50 \equiv 12,400$, while Coral was configured with its optimal concurrency of 5,000 based on Figure 5. As shown in Table 1, P2P-agent generates 2B tokens in 4 hours, achieving $6.8\times$ higher throughput than the official Coral implementation on

the same hardware. Importantly, both systems attain nearly identical agreement correctness, the metric used to measure data quality, consistent with Coral's reported result of 0.456 for LLaMA-3.1-8B-Instruct (Ni et al., 2025a).

5.1.1 Overhead Analysis

We analyze system performance to identify overhead and potential bottlenecks in Matrix. Unless otherwise noted, experiments use 8 H100 nodes (64 GPUs) to generate 200k Coral trajectories.

Latency breakdown. We find that Matrix incurs minimal queuing and orchestration overhead at scale. We instrument end-to-end task latency and attribute it to: (i) agent processing, (ii) queuing delay, and (iii) task initialization. Table 2 reports the breakdown over all trajectories, and Table 3 reports the breakdown for the slowest 10% of trajectories. For typical trajectories, agent processing accounts for ∼80% of end-to-end latency and queuing is negligible. For the slowest trajectories, processing dominates even more (∼99%).

Table 2. Latency breakdown.

Stage	Median	P90	P99
Agent processing	80.12%	99.30%	99.92%
Queuing	0.0289%	0.851%	5.73%
Initialization	0.0051%	1.18%	7.34%

Table 3. Latency breakdown for slow tasks.

Stage	Median	P90	P99
Agent processing	99.72%	99.92%	99.97%
Queuing	0.00172%	0.025%	0.093%
Initialization	0.000005%	0.034%	0.128%

Network bandwidth We estimate the network bandwidth required to transmit orchestration messages. Under the Coral workload, peer-to-peer orchestration generates 2.26M serialized messages and consumes ∼1.6 MB/s of network bandwidth (median ∼1.63 MB/s; P99 ∼3.47 MB/s), indicating that orchestration traffic is modest relative to cluster network capacity.

Bottleneck study To isolate Matrix runtime overhead from model inference cost, we construct dummy Coral agents that do not invoke an LLM and instead return preformatted text by concatenation. We ensure that the synthetic responses match the expected response lengths and turn structure, yielding a "best-case compute" setting that exposes runtime bottlenecks. In this configuration, the system sustains ∼1.1k trajectories/s and processes **12k** orchestration messages per second. The corresponding estimated network bandwidth for orchestration is ∼77.9 MB/s (median ∼82.9 MB/s; P99 ∼97 MB/s). As shown in Table 4, agent processing drops to ∼37% of end-to-end latency, task initialization becomes visible, and queuing remains small. The remaining overhead likely comes from RPC, serialization, and network costs. While this experiment suggests a limit of roughly 12k orchestration messages/s per run, Matrix can exceed this throughput via data parallelism, as discussed in Section 5.2.

Table 4. Latency breakdown of dummy agents without real compute.

Stage	Median	P90	P99
Agent processing	36.72%	62.81%	82.63%
Queuing	0.074%	1.13%	6.95%
Initialization	0.768%	10.72%	24.51%

5.1.2 Actor Crash Recovery

We evaluate robustness by generating 200k Coral trajectories under two settings: (i) no faults, and (ii) injected faults where we randomly kill an agent actor every 12 minutes. Under the at-most-once semantics described in Section 4.5, killing an actor may drop any in-flight orchestrators assigned to it, we therefore report the number of lost tasks. As shown in Table 5, actors are killed 7 times and each time Ray restarts them within seconds on average. Table 6 shows that approximately 2% of tasks are lost in the fault-injection setting, while throughput decreases by only 5%.

Table 5. Coral actors restarts.

Agent	Restarts	Duration	Lost Tasks
answer_extractor	2	0.322	424
answer_matcher	2	0.000	2
student	1	2.304	1474
teacher	2	2.069	2180

Table 6. Impact of agent restarts.

Metric	No Crash	With Crash
Runtime	1:26:16	1:18:43
Total trajectories	200k	200k
Lost trajectories	0	4080
Agreement correctness	0.4781	0.4856
Tokens generated	391,200,916	340,338,986
Tokens per second	75,579	72,059

5.2 NaturalReasoning

NaturalReasoning (Yuan et al., 2025) is a large-scale dataset designed to advance the reasoning capabilities of LLMs

across diverse domains, including STEM, Economics, and Social Sciences. It contains 2.8M challenging questions generated automatically by LLMs. These questions are extracted and synthesized from pretraining corpora, ensuring high diversity and difficulty. Models fine-tuned on NaturalReasoning demonstrate improved sample efficiency and reasoning accuracy compared to prior datasets. In this experiment, we use Matrix to curate a NaturalReasoning-style dataset from raw web documents. This workflow stresses Matrix in a different regime than multi-turn dialogue: most inputs are filtered out early, while the remaining fraction triggers expensive downstream processing. The curation pipeline consists of three agents, as illustrated in Figure 6:

- **Filter:** English-language web documents are identified, and a fine-tuned LLaMA-3.1-3B-Instruct model classifies whether a document contains reasoning content. The classifier is trained on a subset of NaturalReasoning examples as positives and randomly sampled web documents as negatives.
- **Score:** Each document is evaluated along multiple quality axes using LLaMA-3.3-70B-Instruct, following prompts derived from the original NaturalReasoning methodology.
- **Question:** Questions are extracted from the filtered web documents, reference answers are identified when available, and independent reasoning steps leading to a final answer are generated, all using LLaMA-3.3-70B-Instruct. Optionally, we grade the extracted answer and check its consistency against the independently generated answer to further filter low-quality examples.

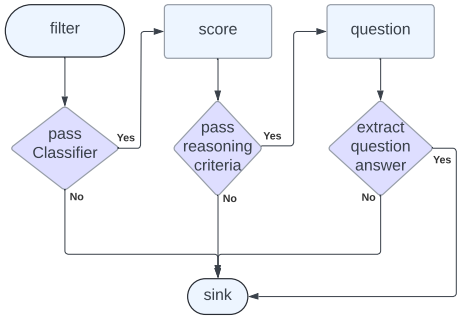

Figure 6. P2P-agents for NaturalReasoning data curation.

For large-scale curation, we process up to 25M web documents from DCLM (Li et al., 2025). The 3B filter model is efficient because most documents are rejected with a single-token (Yes/No) output. Overall, 5.45% of documents pass all filters, yielding approximately 1M high-quality reasoning questions and answers (Table 7).

Table 7. Filtering statistics on 25M DCLM web documents.

Filter step	Percentage
filter_by_en	3.68
filter_by_classifier	90.24
filter_by_score	0.44
filter_by_no_boxed_answer	0.19
success	5.45

5.2.1 *Evaluating Parallelism and Throughput*

Using a 500k DCLM subset, we evaluate the impact of the three types of parallelism supported by Matrix represented as a tuple (data parallelism, task parallelism, and agent parallelism) in Table 8. We deployed 32 A100 nodes with 8 GPUs each. The fine-tuned 3B model was replicated 32 times, while the 70B model used 56 replicas. We set the maximum concurrent tasks to be 14k. The estimated concurrent requests per 70B replica is $14k \times (1 - 3.68\% - 90.24\%) \div 56 \approx 15$, which can maintain high GPU utilization without introducing long latencies or timeouts. The 3B model in Filter agents are not the bottleneck even though they handle 97% of the data after English filter.

Table 8. P2P-agent throughput for 500k webdoc.

Settings Name	Three Parallelisms	Normalized Throughput
1	(1, 14000,1)	1
2	(20, 700, 1)	1.61
3	(240, 1, 1)	0.38
4	(240, 50, 1)	1.43
5	(1, 14000, 2)	1.03
6	(1, 14000, 10)	0.91

Data parallelism. The first two settings present the results for data parallelism. In Setting 1, although the system was configured to allow up to 14k concurrent tasks, only about 700 were observed during the experiment, which is well below the target concurrency. This shortfall occurs because 93% of the input documents are filtered out early (Table 7), so that the input pipeline can not keep up with the Filter agent. To address the input bottleneck, we increased data parallelism by splitting the dataset into 20 partitions for Setting 2. This raises the effective concurrency to $20 \times 700 \equiv 14k$, matching our target. This adjustment yields a $1.61\times$ speedup, demonstrating how data parallelism helps alleviate the input pipeline bottleneck. Increasing the number of partitions beyond 20 provides little additional benefit, since task-level parallelism within each partition already saturates the GPUs.

Task parallelism. Comparing Settings 3 and 4, running 50 concurrent tasks per data partition yields a 3.8× speedup compared to single-task execution, even with 240 data partitions. This result shows that increasing asynchronous task concurrency is more effective than simply creating a larger number of data partitions. Moreover, further increasing data parallelism would require additional agent instances, which in turn demands more CPU resources.

Agent parallelism. Comparing Settings 1 and 5, doubling the number of agent instances (excluding the sink) results in a modest throughput gain; while Setting 6 shows further increasing agent instances has no benefits. This is because LLM inference is handled by Ray Serve, agents remain I/O-bound. While increasing the number of instances offers limited benefit for the NaturalReasoning workflow, Matrix can efficiently scale agent instances when agents perform heavier CPU or GPU computations, highlighting the framework's flexibility and readiness for diverse workloads.

Although the design space of the three kinds of parallelism can be huge, our setup prefers 14k max concurrency given the number of GPUs. We further determined 700 as the maximum achievable asyncio task concurrency per data partition. Moreover, increasing data partitions beyond 20 or increasing agent parallelism beyond 2 has small effect on throughput. Because of the peer-to-peer architecture, task parallelism alone often achieves high resource utilization. Therefore, small degrees of data and agent parallelism are typically sufficient as the initial configuration for new use cases.

5.2.2 *Impact of Scheduling Granularity*

We compare Matrix's *row-level* scheduling to a *batch-level* baseline implemented with Ray Data (Algorithm 2). We emphasize that Ray Data is a general-purpose batch processing engine designed primarily for data-parallel ETL and batched model inference. Our goal in this comparison is *not* to claim that Ray Data is an optimized framework for agentic workflows, but to use it as a representative and widely used batch-oriented alternative for practitioners building scalable LLM-calling pipelines on Ray.

In the Ray Data baseline, each batch is processed by a Ray actor `BatchProcessing` (Lines 1–12), which launches multiple asynchronous tasks to process rows concurrently (Line 8). Each task executes an agentic workflow (Lines 10–12) that is functionally similar to the P2P-agent logic, except that (i) all agents are co-located within the same actor process and (ii) orchestration is implemented within the batch processor rather than being carried by peer-to-peer messages.

This baseline removes the single centralized orchestrator bottleneck and distributes orchestration across many CPU workers, each responsible for one batch. However, because multi-agent workflows have *data-dependent control flow* (e.g., branching, retries, early termination, and variable numbers of steps), conventional batch-inference optimizations are difficult to apply: different rows within a batch may invoke different agents and different numbers of LLM/tool calls. As a result, even under Ray Data, each row must effectively be executed as an independent asynchronous workflow, and the batch mainly serves as a scheduling container rather than enabling true batched execution of LLM calls.

Algorithm 2 Pseudo-code of Ray Data Baseline.

```
1  @ray.remote
2  class BatchProcessing:
3    def __call__(self, batch):
4      async def _process_batch(rows):
5         tasks = [self.process(row) for row in rows]
6         return await asyncio.gather(*tasks)
7
8      return asyncio.run(_process_batch(batch))
9
10   async def process(self, row: Dict[str, Any]):
11     """abstract method to process one input task"""
12     pass
13
14 ds = ray.data.read_json(data_dir)
15 output = ds.map_batches(
16    BatchProcessing,
17    batch_size=cfg.batch_size,
18    num_cpus=1,
19    concurrency=cfg.data_parallelism
20 )
```

Large-Scale Results. We then compare Matrix P2P-agent with the Ray Data baseline to run large scale curation over DCLM up to 25M web documents. Both setups utilize the same GPU resources and 14k concurrent tasks. For the P2P-agent configuration, we adopt Setting 2, i.e., (20, 700, 1), from Table 8. For the Ray Data baseline, we use Setting 4, i.e., (240, 50, 1). Through experiment, Setting 2 with 700 as batch size would result in peaks and valleys in GPU requests, the smaller batch size of 50 in Setting 4 can smooth GPU requests. The two setups have similar throughputs in P2P-agent experiment and the latter fits Ray Data based implementation.

Table 9. P2P-Agent achieves **2.1**× higher token throughput than Ray Data baseline.

Metric	Baseline	P2P-Agent
Runtime	12:57:28	17:57:55
Concurrent tasks	14,000	14,000
Webdoc processed	9.3M	25M
Questions generated	410,755	1,192,799
Tokens generated	129,622,944	378,591,258
Tokens per second	2,778	5,853

Each setup is executed for over 10 hours, measuring token

throughput. Results in Table 9 show that P2P-agent achieves 2.1× higher token throughput than the batch-level baseline. The efficiency gap stems from scheduling granularity: in batch-level scheduling, a new batch cannot begin until all tasks in the current batch complete. Due to control divergence and variable task length, a few slow tasks in a batch block downstream processing, creating idle GPU time. In contrast, row-level scheduling in P2P-agent allows each completed row to immediately trigger the next task without waiting for others, fully utilizing compute resources. Similar behaviour has been observed in LLM inference systems, where "continuous batching" or token-level scheduling can replace completed requests dynamically to avoid idle slots and maintain high throughput.

In Ray Data, decreasing the batch size can partially mitigate idle time. However, each concurrent batch requires a dedicated actor and CPU allocation. Maintaining the same level of task concurrency at smaller batch sizes therefore demands higher data parallelism, which introduces substantial CPU overhead. Moreover, batch-level scheduling incurs additional costs for batch creation and actor management, further compounding inefficiency. Overall, these results demonstrate that fine-grained, row-level scheduling enables more efficient scaling for multi-agent, dynamically controlled workflows than batch-level scheduling in traditional distributed data processing engines.

5.3 Tau2-bench

Tau2-bench (Barres et al., 2025a) is a recently introduced benchmark for evaluating conversational agents in dual-control environments, where both an AI agent and a user simulator interact with a shared environment through tools and APIs. In this experiment, we use Tau2-bench to generate task-solving trajectories for real-world customer support or troubleshooting in the telecom domain. Following prior work such as Kimi K2 (Bai et al., 2025) and AgentBank (Song et al., 2024), these trajectories—after filtering and reward validation—can serve as post-training data to enhance LLM reasoning and tool-use performance.

P2P-Agent Implementation. Matrix implements Tau2-Bench as a distributed P2P-agent workflow comprising four functional agents and one orchestrator (Figure 7).

- **User-simulator:** Represents the human user, initiating and responding to the `tau2-agent`'s queries.
- **Assistant:** Acts as the assistant agent, performing reasoning and tool-use steps.
- **Tool-executor:** Executes HTTP-based tool calls issued by either the user or assistant. Tool APIs are adapted from the official Tau2-agent implementation (Barres et al., 2025b) and deployed in distributed containers to enable concurrent execution and isolation.
- **Reward-calculator:** Validates each trajectory by replaying all tool calls from the initial state and computing task-specific rewards using assertions over the database state. The calculator container reuses the official Tau2-agent implementation, ensuring comparability with benchmark metrics.

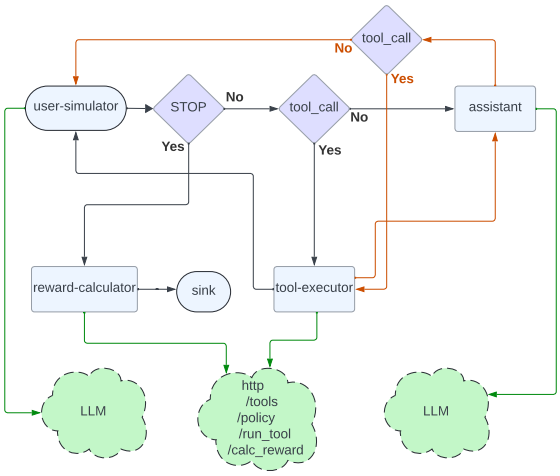

Figure 7. P2P-agent for Tau2-Bench.

Matrix exposes two categories of services: (1) LLM inference services using gpt-oss-120b (OpenAI, 2025), which provide scalable access to model reasoning and dialogue generation, and (2) containerized task services, derived from Tau2-Bench's reference implementation. Each container exposes standardized HTTP endpoints for retrieving tool signatures, executing actions, and evaluating rewards. Service calls are depicted in green in Figure 7.

5.3.1 *Comparison with Tau2 Baseline*

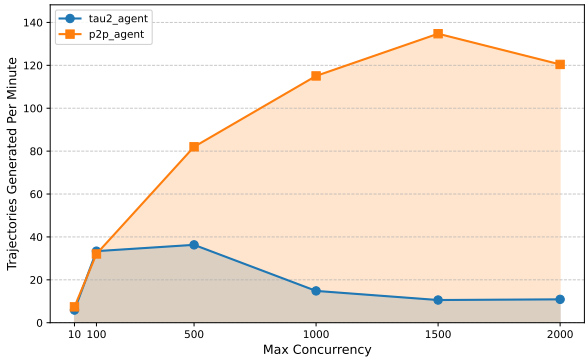

Figure 8. Scalability of P2P-agent vs Tau2-agent baseline.

To evaluate scalability, we compare Matrix's P2P-agent execution with the official Tau2-agent implementation (Barres et al., 2025b). The baseline runs all tools and environment logic directly in Python threads on a single node with distributed LLM service. In contrast, P2P-agent distributes agents, LLM and tool-call container services across the Ray cluster.

As shown in Figure 8, throughput for the Tau2-agent baseline saturates at around 500 threads due to the single-machine constraint. In contrast, P2P-agent continues to scale with concurrency, leveraging distributed placement of agents and containers across the cluster.

Large-Scale Results. We further test on 13 H100 nodes, deploying 1.5k containers and 56 gpt-oss-120b replicas. As shown in Table 10, P2P-agent generates $15.4\times$ more tokens per second than the Tau2-agent baseline, while maintaining comparable task rewards.

Table 10. P2P-Agent achieves **15.4**$\times$ higher token throughput than Tau2-Agent baseline.

Metric	Baseline	P2P-Agent
Runtime	1:13:41	1:15:21
Concurrent tasks	500	1,500
Total trajectories	1519	22,800
Average reward	0.5918	0.5921
Tokens generated	11,080,385	185,376,127
Tokens per second	2,654	41,003

5.3.2 *Effect of Message Offloading*

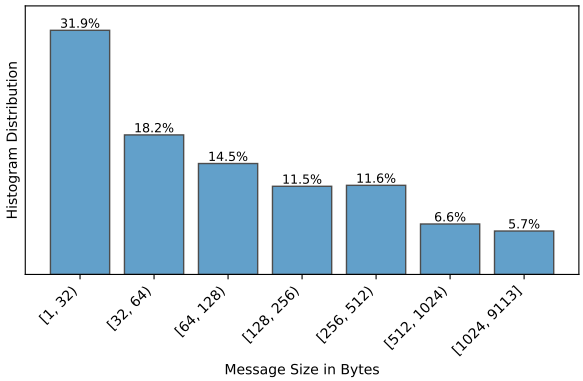

Figure 9. Distribution of conversation sizes in Tau2-Bench (Telecom).

Matrix orchestrator contains the conversation history. Conversations exchanged in P2P-agent Tau2-bench trajectories vary widely in size, as shown in Figure 9. When orchestrators are routed through distributed agents, large conversation content can cause network overhead and congestion within the cluster. To mitigate this overhead, Matrix offloads large conversation content to the Ray Object Store, as discussed in Section 4.6. In this case, contents exceeding 512 bytes are stored in Ray object store and retrieved on demand, which corresponds to about 12% of the conversations.

Figure 10 compares the total cluster network bandwidth during two identical runs: one with message offloading enabled (before 18:30) and one without it (after 18:30). Excluding transient spikes, peak utilization drops from roughly 1 GB/s to 760 MB/s, a reduction of about 20%. This demonstrates that offloading large conversation contents effectively reduces network traffic and improves scalability under communication-heavy workloads such as Tau2-bench. It also makes the system well suited for future multi-modal data generation tasks.

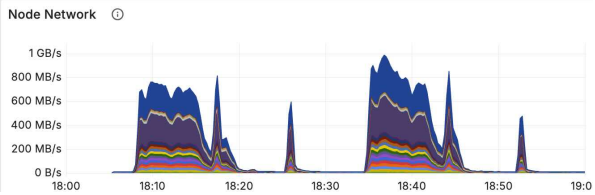

Figure 10. Compare Total Node Network with and without Message Offloading.

6 Conclusion and Discussions

We introduced **Matrix**, a peer-to-peer multi-agent framework for large-scale synthetic data generation. By representing control and data flow as peer-to-peer messages and delegating computation to distributed services, Matrix eliminates centralized bottlenecks and enables efficient execution of tens of thousands of concurrent agent workflows. Matrix is modular and configurable, allowing users to easily adapt it to diverse data generation tasks and agent roles without modifying core logic. We open-source Matrix at `https://github.com/facebookresearch/matrix` to support reproducibility and further research.

Limitations and future work. Matrix is not a universal fit for all multi-agent workloads. It assumes per-task orchestrator state is serializable and can be passed between agents. It is also less suitable when each step must read/write very large mutable shared state, where data movement or synchronization can dominate costs. Finally, Matrix depends on the Ray actor runtime and therefore inherits its operational constraints and failure semantics. Looking forward, we plan to provide a library of reusable orchestrator patterns and end-to-end examples, so users can instantiate common control-flow templates with minimal custom code. Future extensions will also explore multi-modal data generation and on-policy continuous data synthesis.

REFERENCES

Abdin, M., Aneja, J., Behl, H., Bubeck, S., Eldan, R., Gunasekar, S., Harrison, M., Hewett, R. J., Javaheripi, M., Kauffmann, P., Lee, J. R., Lee, Y. T., Li, Y., Liu, W., Mendes, C. C. T., Nguyen, A., Price, E., de Rosa, G., Saarikivi, O., Salim, A., Shah, S., Wang, X., Ward, R., Wu, Y., Yu, D., Zhang, C., and Zhang, Y. Phi-4 technical report, 2024. URL https://arxiv.org/abs/2412.08905.

Bai, Y., Bao, Y., Chen, G., Chen, J., Chen, N., Chen, R., Chen, Y., Chen, Y., Chen, Y., Chen, Z., Cui, J., Ding, H., Dong, M., Du, A., Du, C., Du, D., Du, Y., Fan, Y., Feng, Y., Fu, K., Gao, B., Gao, H., Gao, P., Gao, T., Gu, X., Guan, L., Guo, H., Guo, J., Hu, H., Hao, X., He, T., He, W., He, W., Hong, C., Hu, Y., Hu, Z., Huang, W., Huang, Z., Huang, Z., Jiang, T., Jiang, Z., Jin, X., Kang, Y., Lai, G., Li, C., Li, F., Li, H., Li, M., Li, W., Li, Y., Li, Y., Li, Z., Li, Z., Lin, H., Lin, X., Lin, Z., Liu, C., Liu, C., Liu, H., Liu, J., Liu, J., Liu, L., Liu, S., Liu, T. Y., Liu, T., Liu, W., Liu, Y., Liu, Y., Liu, Y., Liu, Y., Liu, Z., Lu, E., Lu, L., Ma, S., Ma, X., Ma, Y., Mao, S., Mei, J., Men, X., Miao, Y., Pan, S., Peng, Y., Qin, R., Qu, B., Shang, Z., Shi, L., Shi, S., Song, F., Su, J., Su, Z., Sun, X., Sung, F., Tang, H., Tao, J., Teng, Q., Wang, C., Wang, D., Wang, F., Wang, H., Wang, J., Wang, J., Wang, J., Wang, S., Wang, S., Wang, Y., Wang, Y., Wang, Y., Wang, Y., Wang, Y., Wang, Z., Wang, Z., Wang, Z., Wei, C., Wei, Q., Wu, W., Wu, X., Wu, Y., Xiao, C., Xie, X., Xiong, W., Xu, B., Xu, J., Xu, J., Xu, L. H., Xu, L., Xu, S., Xu, W., Xu, X., Xu, Y., Xu, Z., Yan, J., Yan, Y., Yang, X., Yang, Y., Yang, Z., Yang, Z., Yang, Z., Yao, H., Yao, X., Ye, W., Ye, Z., Yin, B., Yu, L., Yuan, E., Yuan, H., Yuan, M., Zhan, H., Zhang, D., Zhang, H., Zhang, W., Zhang, X., Zhang, Y., Zhang, Y., Zhang, Y., Zhang, Y., Zhang, Y., Zhang, Y., Zhang, Z., Zhao, H., Zhao, Y., Zheng, H., Zheng, S., Zhou, J., Zhou, X., Zhou, Z., Zhu, Z., Zhuang, W., and Zu, X. Kimi k2: Open agentic intelligence, 2025. URL https://arxiv.org/abs/2507.20534.

Barrak, A., Jaziri, M., Trabelsi, R., Jaafar, F., and Petrillo, F. Spirt: A fault-tolerant and reliable peer-to-peer serverless ml training architecture, 2023. URL https://arxiv.org/abs/2309.14148.

Barres, V., Dong, H., Ray, S., Si, X., and Narasimhan, K. τ^2-bench: Evaluating conversational agents in a dual-control environment, 2025a. URL https://arxiv.org/abs/2506.07982.

Barres, V., Dong, H., Ray, S., Si, X., and Narasimhan, K. τ^2-bench: Evaluating conversational agents in a dual-control environment, 2025b. URL https://github.com/sierra-research/tau2-bench.

Betker, J., Goh, G., Jing, L., et al. Improving image generation with better captions. Technical report, OpenAI, 2023. URL https://cdn.openai.com/papers/dall-e-3.pdf. Technical report describing DALL·E 3; accessed DATE.

Chan, J. S., Chowdhury, N., Jaffe, O., Aung, J., Sherburn, D., Mays, E., Starace, G., Liu, K., Maksin, L., Patwardhan, T., Weng, L., and Madry, A. Mle-bench: Evaluating machine learning agents on machine learning engineering, 2025. URL https://arxiv.org/abs/2410.07095.

Copet, J., Carbonneaux, Q., Cohen, G., Gehring, J., Kahn, J., Kossen, J., Kreuk, F., McMilin, E., Meyer, M., Wei, Y., Zhang, D., Zheng, K., Armengol-Estapé, J., Bashiri, P., Beck, M., Chambon, P., Charnalia, A., Cummins, C., Decugis, J., Fisches, Z. V., Fleuret, F., Gloeckle, F., Gu, A., Hassid, M., Haziza, D., Idrissi, B. Y., Keller, C., Kindi, R., Leather, H., Maimon, G., Markosyan, A., Massa, F., Mazaré, P.-E., Mella, V., Murray, N., Muzumdar, K., O'Hearn, P., Pagliardini, M., Pedchenko, D., Remez, T., Seeker, V., Selvi, M., Sultan, O., Wang, S., Wehrstedt, L., Yoran, O., Zhang, L., Cohen, T., Adi, Y., and Synnaeve, G. Cwm: An open-weights llm for research on code generation with world models, 2025. URL https://arxiv.org/abs/2510.02387.

CrewAI. Crewai: Open-source multi-agent framework for collaborative artificial intelligence. https://www.crewai.com, 2025. Accessed: 2025-10-22.

FAIR, M. Fastgen: Simple high-throughput inference library. https://github.com/facebookresearch/fastgen, 2025. Accessed: 2025-10-24.

Fourney, A., Bansal, G., Mozannar, H., Tan, C., Salinas, E., Erkang, Zhu, Niedtner, F., Proebsting, G., Bassman, G., Gerrits, J., Alber, J., Chang, P., Loynd, R., West, R., Dibia, V., Awadallah, A., Kamar, E., Hosn, R., and Amershi, S. Magentic-one: A generalist multi-agent system for solving complex tasks, 2024. URL https://arxiv.org/abs/2411.04468.

Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Vaughan, A., Yang, A., Fan, A., Goyal, A., Hartshorn, A., Yang, A., Mitra, A., Sravankumar, A., Korenev, A., Hinsvark, A., Rao, A., Zhang, A., Rodriguez, A., Gregerson, A., Spataru, A., Roziere, B., Biron, B., Tang, B., Chern, B., Caucheteux, C., Nayak, C., Bi, C., Marra, C., McConnell, C., Keller, C., Touret, C., Wu, C., Wong, C., Ferrer, C. C., Nikolaidis, C., Allonsius, D., Song, D., Pintz, D., Livshits, D., Wyatt, D., Esiobu, D., Choudhary, D., Mahajan, D., Garcia-Olano, D., Perino, D., Hupkes, D., Lakomkin, E., AlBadawy, E., Lobanova, E., Dinan,

E., Smith, E. M., Radenovic, F., Guzmán, F., Zhang, F., Synnaeve, G., Lee, G., Anderson, G. L., Thattai, G., Nail, G., Mialon, G., Pang, G., Cucurell, G., Nguyen, H., Korevaar, H., Xu, H., Touvron, H., Zarov, I., Ibarra, I. A., Kloumann, I., Misra, I., Evtimov, I., Zhang, J., Copet, J., Lee, J., Geffert, J., Vranes, J., Park, J., Mahadeokar, J., Shah, J., van der Linde, J., Billock, J., Hong, J., Lee, J., Fu, J., Chi, J., Huang, J., Liu, J., Wang, J., Yu, J., Bitton, J., Spisak, J., Park, J., Rocca, J., Johnstun, J., Saxe, J., Jia, J., Alwala, K. V., Prasad, K., Upasani, K., Plawiak, K., Li, K., Heafield, K., Stone, K., El-Arini, K., Iyer, K., Malik, K., Chiu, K., Bhalla, K., Lakhotia, K., Rantala-Yeary, L., van der Maaten, L., Chen, L., Tan, L., Jenkins, L., Martin, L., Madaan, L., Malo, L., Blecher, L., Landzaat, L., de Oliveira, L., Muzzi, M., Pasupuleti, M., Singh, M., Paluri, M., Kardas, M., Tsimpoukelli, M., Oldham, M., Rita, M., Pavlova, M., Kambadur, M., Lewis, M., Si, M., Singh, M. K., Hassan, M., Goyal, N., Torabi, N., Bashlykov, N., Bogoychev, N., Chatterji, N., Zhang, N., Duchenne, O., Çelebi, O., Alrassy, P., Zhang, P., Li, P., Vasic, P., Weng, P., Bhargava, P., Dubal, P., Krishnan, P., Koura, P. S., Xu, P., He, Q., Dong, Q., Srinivasan, R., Ganapathy, R., Calderer, R., Cabral, R. S., Stojnic, R., Raileanu, R., Maheswari, R., Girdhar, R., Patel, R., Sauvestre, R., Polidoro, R., Sumbaly, R., Taylor, R., Silva, R., Hou, R., Wang, R., Hosseini, S., Chennabasappa, S., Singh, S., Bell, S., Kim, S. S., Edunov, S., Nie, S., Narang, S., Raparthy, S., Shen, S., Wan, S., Bhosale, S., Zhang, S., Vandenhende, S., Batra, S., Whitman, S., Sootla, S., Collot, S., Gururangan, S., Borodinsky, S., Herman, T., Fowler, T., Sheasha, T., Georgiou, T., Scialom, T., Speckbacher, T., Mihaylov, T., Xiao, T., Karn, U., Goswami, V., Gupta, V., Ramanathan, V., Kerkez, V., Gonguet, V., Do, V., Vogeti, V., Albiero, V., Petrovic, V., Chu, W., Xiong, W., Fu, W., Meers, W., Martinet, X., Wang, X., Wang, X., Tan, X. E., Xia, X., Xie, X., Jia, X., Wang, X., Goldschlag, Y., Gaur, Y., Babaei, Y., Wen, Y., Song, Y., Zhang, Y., Li, Y., Mao, Y., Coudert, Z. D., Yan, Z., Chen, Z., Papakipos, Z., Singh, A., Srivastava, A., Jain, A., Kelsey, A., Shajnfeld, A., Gangidi, A., Victoria, A., Goldstand, A., Menon, A., Sharma, A., Boesenberg, A., Baevski, A., Feinstein, A., Kallet, A., Sangani, A., Teo, A., Yunus, A., Lupu, A., Alvarado, A., Caples, A., Gu, A., Ho, A., Poulton, A., Ryan, A., Ramchandani, A., Dong, A., Franco, A., Goyal, A., Saraf, A., Chowdhury, A., Gabriel, A., Bharambe, A., Eisenman, A., Yazdan, A., James, B., Maurer, B., Leonhardi, B., Huang, B., Loyd, B., Paola, B. D., Paranjape, B., Liu, B., Wu, B., Ni, B., Hancock, B., Wasti, B., Spence, B., Stojkovic, B., Gamido, B., Montalvo, B., Parker, C., Burton, C., Mejia, C., Liu, C., Wang, C., Kim, C., Zhou, C., Hu, C., Chu, C.-H., Cai, C., Tindal, C., Feichtenhofer, C., Gao, C., Civin, D., Beaty, D., Kreymer, D., Li, D., Adkins, D., Xu, D., Testuggine, D., David, D., Parikh, D., Liskovich, D., Foss, D., Wang, D., Le, D., Holland, D., Dowling, E., Jamil, E., Montgomery, E., Presani, E., Hahn, E., Wood, E., Le, E.-T., Brinkman, E., Arcaute, E., Dunbar, E., Smothers, E., Sun, F., Kreuk, F., Tian, F., Kokkinos, F., Ozgenel, F., Caggioni, F., Kanayet, F., Seide, F., Florez, G. M., Schwarz, G., Badeer, G., Swee, G., Halpern, G., Herman, G., Sizov, G., Guangyi, Zhang, Lakshminarayanan, G., Inan, H., Shojanazeri, H., Zou, H., Wang, H., Zha, H., Habeeb, H., Rudolph, H., Suk, H., Aspegren, H., Goldman, H., Zhan, H., Damlaj, I., Molybog, I., Tufanov, I., Leontiadis, I., Veliche, I.-E., Gat, I., Weissman, J., Geboski, J., Kohli, J., Lam, J., Asher, J., Gaya, J.-B., Marcus, J., Tang, J., Chan, J., Zhen, J., Reizenstein, J., Teboul, J., Zhong, J., Jin, J., Yang, J., Cummings, J., Carvill, J., Shepard, J., McPhie, J., Torres, J., Ginsburg, J., Wang, J., Wu, K., U, K. H., Saxena, K., Khandelwal, K., Zand, K., Matosich, K., Veeraraghavan, K., Michelena, K., Li, K., Jagadeesh, K., Huang, K., Chawla, K., Huang, K., Chen, L., Garg, L., A, L., Silva, L., Bell, L., Zhang, L., Guo, L., Yu, L., Moshkovich, L., Wehrstedt, L., Khabsa, M., Avalani, M., Bhatt, M., Mankus, M., Hasson, M., Lennie, M., Reso, M., Groshev, M., Naumov, M., Lathi, M., Keneally, M., Liu, M., Seltzer, M. L., Valko, M., Restrepo, M., Patel, M., Vyatskov, M., Samvelyan, M., Clark, M., Macey, M., Wang, M., Hermoso, M. J., Metanat, M., Rastegari, M., Bansal, M., Santhanam, N., Parks, N., White, N., Bawa, N., Singhal, N., Egebo, N., Usunier, N., Mehta, N., Laptev, N. P., Dong, N., Cheng, N., Chernoguz, O., Hart, O., Salpekar, O., Kalinli, O., Kent, P., Parekh, P., Saab, P., Balaji, P., Rittner, P., Bontrager, P., Roux, P., Dollar, P., Zvyagina, P., Ratanchandani, P., Yuvraj, P., Liang, Q., Alao, R., Rodriguez, R., Ayub, R., Murthy, R., Nayani, R., Mitra, R., Parthasarathy, R., Li, R., Hogan, R., Battey, R., Wang, R., Howes, R., Rinott, R., Mehta, S., Siby, S., Bondu, S. J., Datta, S., Chugh, S., Hunt, S., Dhillon, S., Sidorov, S., Pan, S., Mahajan, S., Verma, S., Yamamoto, S., Ramaswamy, S., Lindsay, S., Lindsay, S., Feng, S., Lin, S., Zha, S. C., Patil, S., Shankar, S., Zhang, S., Zhang, S., Wang, S., Agarwal, S., Sajuyigbe, S., Chintala, S., Max, S., Chen, S., Kehoe, S., Satterfield, S., Govindaprasad, S., Gupta, S., Deng, S., Cho, S., Virk, S., Subramanian, S., Choudhury, S., Goldman, S., Remez, T., Glaser, T., Best, T., Koehler, T., Robinson, T., Li, T., Zhang, T., Matthews, T., Chou, T., Shaked, T., Vontimitta, V., Ajayi, V., Montanez, V., Mohan, V., Kumar, V. S., Mangla, V., Ionescu, V., Poenaru, V., Mihailescu, V. T., Ivanov, V., Li, W., Wang, W., Jiang, W., Bouaziz, W., Constable, W., Tang, X., Wu, X., Wang, X., Wu, X., Gao, X., Kleinman, Y., Chen, Y., Hu, Y., Jia, Y., Qi, Y., Li, Y., Zhang, Y., Zhang, Y., Adi, Y., Nam, Y., Yu, Wang, Zhao, Y., Hao, Y., Qian, Y., Li, Y., He, Y., Rait, Z., DeVito, Z., Rosnbrick, Z., Wen, Z., Yang, Z., Zhao, Z., and Ma, Z. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., and Steinhardt, J. Measuring mathematical problem solving with the math dataset. *NeurIPS*, 2021.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

Kurtzer, G. M., Sochat, V., and Bauer, M. W. Singularity: Scientific containers for mobility of compute. *PLoS ONE*, 12(5):e0177459, 2017. doi: 10.1371/journal.pone.0177459. URL https://doi.org/10.1371/journal.pone.0177459.

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

Labs, G. Grafana: The open and composable observability and data visualization platform. https://github.com/grafana/grafana, 2025. Accessed: 2025-10-24.

LangChain. Langgraph: A low-level orchestration framework for stateful ai agents. https://langchain-ai.github.io/langgraph/, 2025. Accessed: 2025-10-22.

Li, J., Fang, A., Smyrnis, G., Ivgi, M., Jordan, M., Gadre, S., Bansal, H., Guha, E., Keh, S., Arora, K., Garg, S., Xin, R., Muennighoff, N., Heckel, R., Mercat, J., Chen, M., Gururangan, S., Wortsman, M., Albalak, A., Bitton, Y., Nezhurina, M., Abbas, A., Hsieh, C.-Y., Ghosh, D., Gardner, J., Kilian, M., Zhang, H., Shao, R., Pratt, S., Sanyal, S., Ilharco, G., Daras, G., Marathe, K., Gokaslan, A., Zhang, J., Chandu, K., Nguyen, T., Vasiljevic, I., Kakade, S., Song, S., Sanghavi, S., Faghri, F., Oh, S., Zettlemoyer, L., Lo, K., El-Nouby, A., Pouransari, H., Toshev, A., Wang, S., Groeneveld, D., Soldaini, L., Koh, P. W., Jitsev, J., Kollar, T., Dimakis, A. G., Carmon, Y., Dave, A., Schmidt, L., and Shankar, V. Datacomp-lm: In search of the next generation of training sets for language models, 2025. URL https://arxiv.org/abs/2406.11794.

Mitra, A., Corro, L. D., Zheng, G., Mahajan, S., Rouhana, D., Codas, A., Lu, Y., ge Chen, W., Vrousgos, O., Rosset, C., Silva, F., Khanpour, H., Lara, Y., and Awadallah, A. Agentinstruct: Toward generative teaching with agentic flows, 2024. URL https://arxiv.org/abs/2407.03502.

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., and Stoica, I. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 561–577, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/moritz.

Ni, A., Desai, R., Li, Y., Lei, X., Wang, D., Zhang, J., Yu, J., Raghavendra, R., Ghosh, G., Li, S.-W., and Celikyilmaz, A. Collaborative reasoner: Self-improving social agents with synthetic conversations. In *NeurIPS 2025*, 2025a.

Ni, A., Desai, R., Li, Y., Lei, X., Wang, D., Zhang, J., Yu, J., Raghavendra, R., Ghosh, G., Li, S.-W., and Celikyilmaz, A. Collaborative reasoner: Self-improving social agents with synthetic conversations, 2025b. URL https://github.com/facebookresearch/collaborative-reasoner.

OpenAI. gpt-oss-120b & gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

Pham, M. V. T., Phan, H. N., Phan, H. N., Chi, C. L., Nguyen, T. N., and Bui, N. D. Q. Swe-synth: Synthesizing verifiable bug-fix data to enable large language models in resolving real-world bugs, 2025. URL https://arxiv.org/abs/2504.14757.

Prabhakar, A., Liu, Z., Zhu, M., Zhang, J., Awalgaonkar, T., Wang, S., Liu, Z., Chen, H., Hoang, T., Niebles, J. C., Heinecke, S., Yao, W., Wang, H., Savarese, S., and Xiong, C. Apigen-mt: Agentic pipeline for multi-turn data generation via simulated agent-human interplay, 2025. URL https://arxiv.org/abs/2504.03601.

Qin, Z., Yan, X., Zhou, M., and Deng, S. Blockdfl: A blockchain-based fully decentralized peer-to-peer federated learning framework, 2024. URL https://arxiv.org/abs/2205.10568.

Shi, D., Cao, J., Chen, Q., Sun, W., Li, W., Lu, H., Dong, F., Qin, T., Zhu, K., Liu, M., Yang, J., Zhang, G., Liu, J., Zhang, C., Wang, J., Jiang, Y. E., and Zhou, W. Taskcraft: Automated generation of agentic tasks, 2025. URL https://arxiv.org/abs/2506.10055.

Song, Y., Xiong, W., Zhao, X., Zhu, D., Wu, W., Wang, K., Li, C., Peng, W., and Li, S. Agentbank: Towards generalized llm agents via fine-tuning on 50000+ interaction trajectories, 2024. URL https://arxiv.org/abs/2410.07706.

Wang, Y., Ma, X., Zhang, G., Ni, Y., Chandra, A., Guo, S., Ren, W., Arulraj, A., He, X., Jiang, Z., Li, T., Ku,

M., Wang, K., Zhuang, A., Fan, R., Yue, X., and Chen, W. Mmlu-pro: A more robust and challenging multi-task language understanding benchmark, 2024. URL https://arxiv.org/abs/2406.01574.

Wang, Z., Chang, Q., Patel, H., Biju, S., Wu, C.-E., Liu, Q., Ding, A., Rezazadeh, A., Shah, A., Bao, Y., and Siow, E. Mcp-bench: Benchmarking tool-using llm agents with complex real-world tasks via mcp servers, 2025. URL https://arxiv.org/abs/2508.20453.

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Liu, J., Awadallah, A. H., White, R. W., Burger, D., and Wang, C. Autogen: Enabling next-gen llm applications via multi-agent conversation, 2023. URL https://arxiv.org/abs/2308.08155.

Xie, J., Xu, D., Zhao, X., and Song, D. Agentsynth: Scalable task generation for generalist computer-use agents, 2025. URL https://arxiv.org/abs/2506.14205.

Yadan, O. Hydra - a framework for elegantly configuring complex applications. Github, 2019. URL https://github.com/facebookresearch/hydra.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024. URL https://arxiv.org/abs/2405.15793.

Yoo, A. B., Jette, M. A., and Grondona, M. Slurm: Simple linux utility for resource management. In Feitelson, D., Rudolph, L., and Schwiegelshohn, U. (eds.), *Job Scheduling Strategies for Parallel Processing*, pp. 44–60, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-39727-4.

Yuan, W., Yu, J., Jiang, S., Padthe, K., Li, Y., Kulikov, I., Cho, K., Wang, D., Tian, Y., Weston, J. E., and Li, X. Naturalreasoning: Reasoning in the wild with 2.8m challenging questions, 2025. URL https://arxiv.org/abs/2502.13124.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, 2012.

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.