Meta

# MATRIX

## Peer-to-Peer Multi-Agent Synthetic Data Generation Framework

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

Meta FAIR

MLSYS 2026

# The Importance of Synthetic Data

Large language models are increasingly trained with synthetic data to supplement human-curated datasets, reducing dependence on costly, noisy, or privacy-sensitive sources. Recent advances highlight the value of **agentic synthetic data generation**, where complex data is produced through interactions among multiple intelligent agents rather than a single model or fixed pipeline.

State-of-the-Art Examples:

## Kimi K2

Employs a large-scale multi-agent data synthesis pipeline to construct diverse tool-use and reasoning demonstrations.

## Meta CWM

Leverages autonomous software engineering agents to generate multi-step trajectories for code understanding and debugging.

# Related Work

## Generic Frameworks

- Ray: Foundational distributed computing, but lacks native multi-agent abstractions.
- LangChain / LangGraph: Excellent for orchestrating LLM calls, but often runs sequentially on a single node.
- AutoGen: Expressive multi-agent conversation framework optimized for workflow authoring, with additional engineering needed for production-scale distributed execution.

## Synthetic Data Frameworks

- AgentInstruct: Multi-turn instruction-response data via multiple agents.
- TaskCraft: Multi-step, multi-tool agentic tasks with verifiable trajectories.
- APIGen-MT: Two-phase framework for verifiable agent interaction data.
- SWE-Agent & SWE-Synth: Automated software engineering and bug-fix data.

### The Missing Middle

Existing solutions force a trade-off: Generic frameworks require **heavy engineering** to build scalable multi-agent systems from scratch, while Synthetic Data frameworks are **domain-specific** and need adaptation for new tasks. Matrix bridges this gap by providing a scalable, general-purpose runtime specifically designed for multi-agent data generation.

# The Demands of Modern Synthetic Data

## Control Intensive

Modern workflows require complex **multi-agent orchestration**. Generating high-quality data involves multiple steps of reasoning, tool use, and verification that cannot be handled by a simple linear pipeline.

## Diverse Needs

Systems must be highly **configurable and flexible**. Researchers need to easily swap models, tools, and control flows without rebuilding infrastructure for every new task.

## The Efficiency Requirement

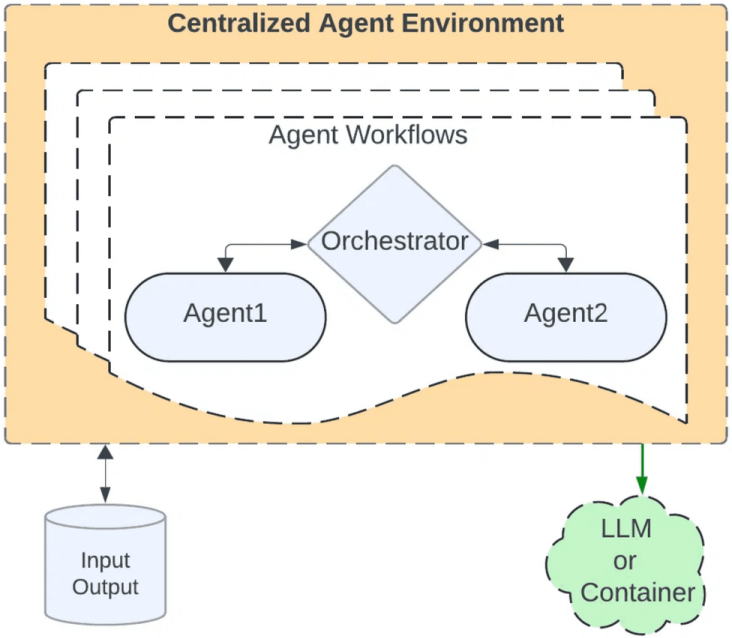
High throughput is a fundamental requirement driven by the **"Generate & Reject"** paradigm:

- **Data Source Filtering:** E.g., NaturalReasoning filters millions of web documents, discarding the vast majority.
- **Rejection Sampling:** Generating massive candidate pools to find a few high-quality trajectories.

This requires a system that can process tens of thousands of concurrent tasks without orchestrator bottlenecks.

# Centralized Orchestration Creates a Bottleneck

- **Centralized Orchestrator:** Acts as a single point of failure and creates a severe bottleneck at high concurrency when scheduling thousands of tasks.
- **Batch-Level Scheduling:** Long-running tasks stall the entire batch, leaving downstream GPUs idle and reducing overall cluster utilization.
- **Hardcoded Domain Logic:** Frameworks are often optimized around specific task structures, making adaptation difficult.
- **Per-Process Concurrency:** Requires external job schedulers and manual provisioning of shared services for inference and tools.



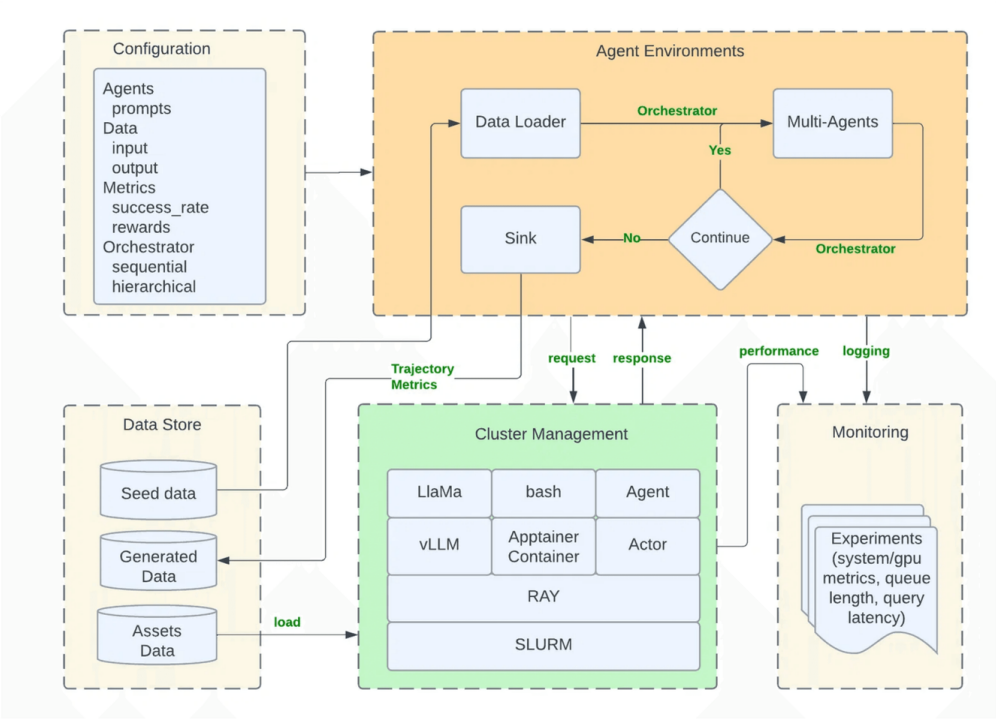
# Introducing Matrix

**Matrix** is a scalable runtime for large-scale multi-agent synthetic data generation capable of efficiently executing tens of thousands of concurrent workflows. It adopts a **peer-to-peer agent architecture** with message-embedded control and state representation.

### Configurable & Flexible: Supporting Diverse Use Cases in One System

- **Multi-agent collaborative dialogue** (e.g., Coral)
- **Web-based reasoning data extraction** (e.g., NaturalReasoning)
- **Tool-use trajectory generation** in customer service environments (e.g., Tau2-Bench)

# Matrix Architecture & Key Properties




### 1 Independent Scaling

Each agent role is a stateless Ray actor pool, allowing them to be **scaled independently** based on specific compute requirements.

### 2 High Task Concurrency

Concurrency is decoupled from actor count. **Tens of thousands of workflows** can flow through a relatively small set of persistent actors.

### 3 Backpressure Control

A global semaphore limits in-flight workflows. The driver admits a new workflow only by **acquiring a token**, preventing system overload.

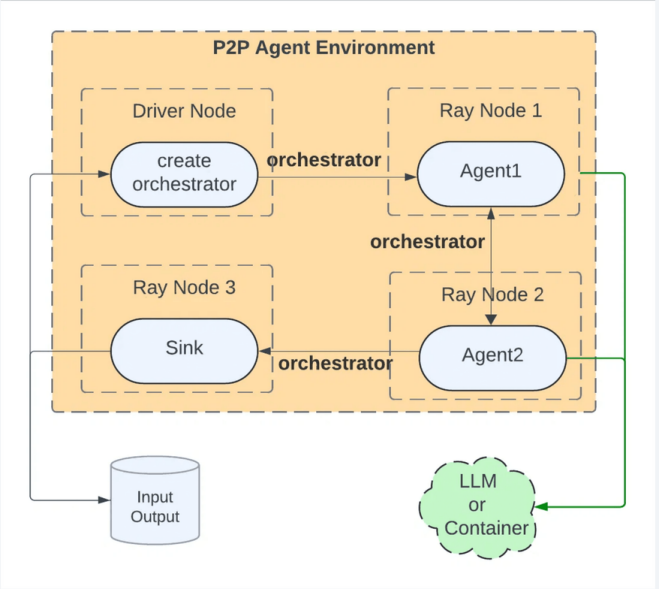
### 4 Modularity

The peer-to-peer design inherently promotes **stateless agents**, serializable orchestrators, and distributed services.

# Peer-to-Peer Agent Orchestration

Matrix replaces the centralized orchestrator with a decentralized, peer-to-peer network:

- **Decentralized Network:** Agents act as independent nodes in a peer-to-peer network, eliminating the need for a central orchestrator.
- **Asynchronous Execution:** Each agent independently pulls tasks, processes them, and routes the output directly to the next agent or sink.
- **Direct Communication:** Messages are passed directly between agents, significantly reducing scheduling overhead and network bottlenecks.



# Reliability & Observability

## Fault Tolerance

Provides at-most-once execution semantic. Each agent role has a broker to track in-flight messages. Broker detects agent crash and mark the in-flight messages as failures. When an agent crashes, it restarts within seconds via Ray Actor. In experiments under fault injection, this resulted in only a **~2% task loss** and a minimal 5% drop in throughput.

## Debuggability

Ray streams actor logs back to the driver process, enabling a “local-like” debugging experience. For failures like timeout, the trajectory includes the relevant error context for offline analysis. Unhandled exceptions propagate to the event loop and serialized to output.

## Monitoring

Deep integration with **Grafana** for real-time observability. Dashboards expose custom indicators like distributed queue length, pending tasks, and query latency to quickly identify bottlenecks.

# Multi-Agent Collaborative Reasoning

- **What is Coral?** The Collaborative Reasoner (Coral) evaluates and improves collaborative reasoning in LLMs through dialogue-driven tasks
- **The Workflow:** Two agents (a "teacher" and a "student") discuss, disagree, and eventually reach consensus over multi-turn interactions.
- **The Matrix Advantage:** Matrix processed 1 Million trajectories in just 4.5 hours (on 31 A100 nodes), achieving **6.8× higher token throughput** than the official centralized baseline.

*Reference: Ni et al., "Collaborative Reasoner: Self-improving social agents with synthetic conversations" (NeurIPS 2025). We use Matrix to implement their system.*

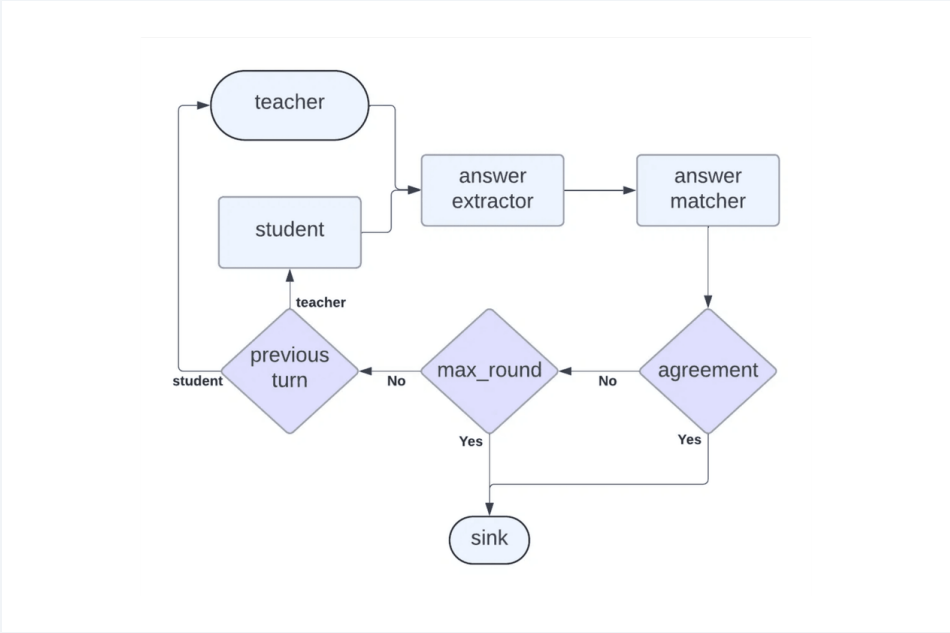


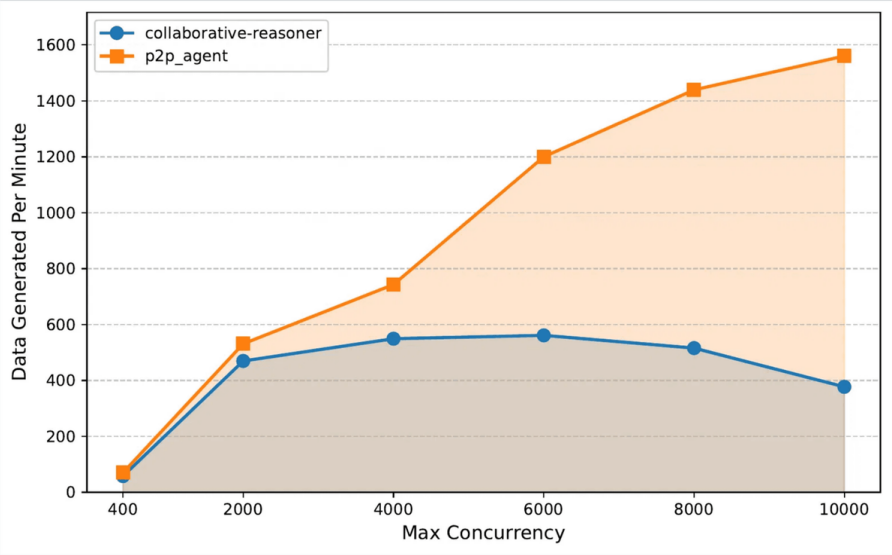

Figure 4: P2P-agents for Collaborative Reasoner

# Matrix Scales Linearly; Centralized Orchestration Hits a Ceiling

The scalability experiment runs both systems on increasing numbers of A100 nodes (400 to 10,000 max concurrency), with 50 concurrent queries per GPU.

**Matrix (P2P-agent):** Throughput scales almost linearly with added GPU nodes. Decentralized coordination means no single bottleneck—adding more nodes directly translates to more data generated.

**Coral (Centralized):** Throughput plateaus around 4,000–6,000 max concurrency due to the orchestrator becoming the scheduling bottleneck.



# Overhead Analysis: Matrix Incurs Minimal Orchestration Cost

We analyzed system performance using 8 H100 nodes to generate 200k Coral trajectories, instrumenting end-to-end task latency.

- **Compute Dominates:** Agent processing accounts for ~80% of end-to-end latency for typical tasks, and ~99% for the slowest tasks.
- **Negligible Overhead:** Queuing and initialization delays remain minimal across all percentiles.
- **High Capacity:** A bottleneck study using dummy agents (no real compute) sustains ~1.1k trajectories/s and processes **12k orchestration messages/s.**
- **Modest Network:** Peer-to-peer orchestration consumes only ~1.6 MB/s of network bandwidth.

Table 2. Latency breakdown (all trajectories)

| 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% |

Table 4. Latency breakdown of dummy agents (no 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% |

# Actor Crash Recovery: Only 2% Task Loss Under Fault Injection

We evaluated robustness by generating 200k Coral trajectories under two settings: (i) no faults, and (ii) injected faults where we randomly kill an agent actor every 10 minutes.

- **Fast Recovery:** Actors are killed 7 times in total; Ray restarts them within seconds on average.
- **Minimal Impact:** Only ~2% of tasks are lost in the fault-injection setting due to at-most-once semantics.
- **Stable Throughput:** Throughput decreases by only 5% (from 75,579 to 72,059 tokens/s).
- **Consistent Quality:** Agreement correctness metric remains stable (0.4781 vs 0.4856).

Table 5. Coral actors restarts

| Agent | Restarts | Duration (s) | 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 |

# Web-Based Reasoning Data Curation

NaturalReasoning curates reasoning datasets from raw web documents. The pipeline consists of three specialized agents:

- **Filter Agent:** Identifies English-language documents and uses a fine-tuned LLaMA-3.1-3B model to classify if the text contains reasoning content.
- **Score Agent:** Evaluates document quality along multiple axes using LLaMA-3.3-70B-Instruct.
- **Question Agent:** Extracts questions, reference answers, and independent reasoning steps.

This workflow stresses Matrix in a high-filtering regime, where most inputs are filtered out early and only a fraction trigger expensive downstream processing.

*Reference: Yuan et al., "NaturalReasoning: Reasoning in the wild with 2.8M challenging questions" (2025).*

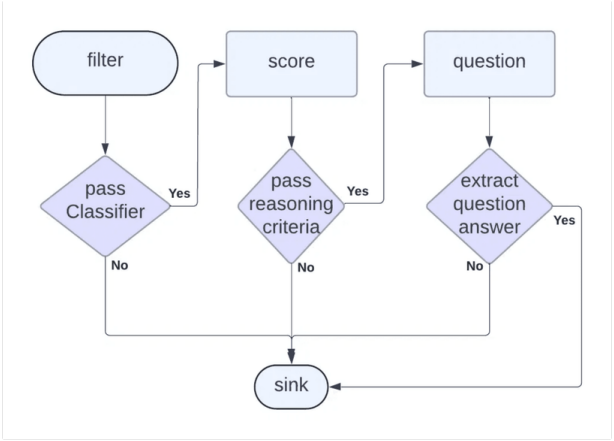



Figure 6: P2P-agents for NaturalReasoning data curation

# Impact of Scheduling Granularity: Row vs Batch

- **Batch-Level (Ray Data Baseline):** A new batch cannot begin until *all* tasks in the current batch complete. Due to variable task lengths and control flow, slow tasks create idle GPU time. The batch is merely a scheduling container, not true batched LLM execution.
- **Row-Level (Matrix):** Each completed row immediately triggers the next task without waiting, fully utilizing compute resources.

Algorithm 2: Pseudo-code of Ray Data Baseline

```
@ray.remote
class BatchProcessing:
    def __call__(self, batch):
        async def _process_batch(rows):
            tasks = [self.process(row) for row in rows]
            return await asyncio.gather(*tasks)
        return asyncio.run(_process_batch(batch))

ds = ray.data.read_json(data_dir)
output = ds.map_batches(
    BatchProcessing,
    batch_size=cfg.batch_size,
    concurrency=cfg.data_parallelism
)
```

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

| Metric | Ray Data Baseline | Matrix 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** |

*Both setups utilized the same GPU resources (32 A100 nodes) and 14k concurrent tasks. Matrix processes 2.7× more documents in comparable time.

# Three Levels of Parallelism

To achieve high throughput in complex, multi-agent workflows like NaturalReasoning, Matrix inherently supports parallelism at three distinct levels.
This multi-tiered approach ensures that both compute-heavy tasks (like LLM inference) and control-heavy tasks (like agent routing) scale efficiently without blocking each other.

## 1. Data Parallelism

Multiple independent data items (workflows) are processed concurrently across the cluster, maximizing overall system utilization.

## 2. Task Parallelism

Different agents (e.g., Filter, Score, Question) operate asynchronously. While one agent processes item A, another agent simultaneously processes item B.

## 3. Agent Parallelism

Each agent Roles can scale horizontally by launching multiple distributed agent instances.

# Parallelism Unlocks Throughput Gains

| Settings Name | Three Parallelisms | Normalized Throughput |
|---|---|---|
| Baseline | (1, 14000, 1) | 1 |
| Increase Data Parallelism | (20, 700, 1) | 1.61 |
| Only Data Parallelism | (240, 1, 1) | 0.38 |
| Increase Task Parallelism | (240, 50, 1) | 1.43 |
| Increase Agent Parallelism | (1, 14000, 2) | 1.03 |
| More Agent Parallelism | (1, 14000, 10) | 0.91 |

*P2P-agent throughput for 500k webdoc. The Three Parallelisms represent (Data Parallelism, Task Parallelism, Agent Parallelism). Splitting the dataset into 20 partitions yields a 1.61× speedup.*

# Multi-Agent Tool-Use Trajectories

Tau2-bench evaluates conversational agents in dual-control environments, where both an AI agent and a user simulator interact with shared tools and APIs.

Matrix implements this complex workflow using a flexible orchestrator comprising four functional agents:

- **User-simulator:** Represents the human user.
- **Assistant:** Performs reasoning and tool-use steps.
- **Tool-executor:** Executes HTTP-based tool calls.
- **Reward-calculator:** Validates trajectories by replaying tool calls and computing task-specific rewards.

*Reference: Barres et al., "τ²-bench: Evaluating conversational agents in a dual-control environment" (2025). We use Matrix to implement their system.*

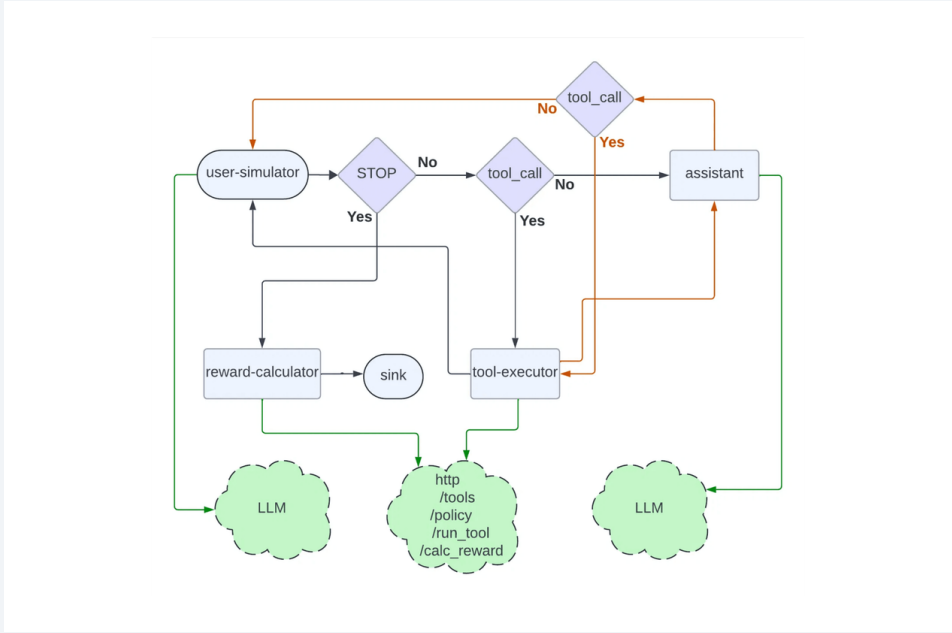



Figure 7: P2P-agent orchestrator graph for Tau2-Bench

# Two Ways to Author Orchestrator

```
async def update(self, result, updater, logger):
    if stop:
        self._current_agent = "remote_reward"
        return self

    if is_tool_call:
        self._current_agent = "remote_env"
    elif self._current_agent == "user_simulator":
        self._current_agent = "llm_agent"
    elif self._current_agent == "llm_agent":
        self._current_agent = "user_simulator"
    elif self._current_agent == "remote_env":
        self._current_agent = last_non_env_agent

    return self
```

```
def build_tau2_graph():
    graph = StateGraph(Tau2State)
    graph.set_entry_point("user_simulator")

    graph.add_conditional_edges(
        "user_simulator",
        lambda s: (
            "remote_reward" if s["should_stop"]
            else "remote_env" if s["has_tool_calls"]
            else "llm_agent"
        )
    )

    graph.add_conditional_edges(
        "remote_env",
        lambda s: ("remote_reward" if s["should_stop"]
                   else s["last_non_env_agent"])
    )

    graph.add_edge("remote_reward", END)
    return graph.compile()
```

# Matrix Flexibility: Containerized Execution

## Apptainer / Docker Support

Matrix natively supports containerized execution via Apptainer (Docker).

This allows complex, stateful, or potentially unsafe tool-use environments to run securely and reproducibly across the distributed cluster without manual environment setup on each node.

## Reusing Official Infrastructure

For Tau2-Bench, Matrix does not reinvent the wheel.

- The **tool-executor** and **reward-calculator** agents run directly inside the official Tau2-bench Docker container.
- This ensures perfect reproducibility for HTTP tool calls, database validation, and reward calculation.

# Mitigating Network Congestion via Message Offloading

In complex environments like Tau2-bench, the conversation history exchanged between agents varies widely in size. Routing this large content through distributed agents can cause network overhead.

- **The Solution:** Matrix offloads large conversation content (> 512 bytes) to the Ray Object Store.
- **Targeted Impact:** Only ~12% of conversations exceed this threshold and require offloading.
- **Network Reduction:** This simple mechanism reduces peak network utilization by roughly 20% (from ~1 GB/s down to ~760 MB/s).
- **Future Proof:** Effectively prepares the system for communication-heavy workloads, such as multi-modal data generation.

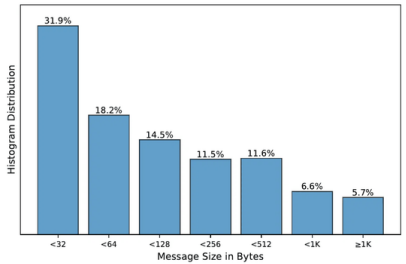



Figure 9: Distribution of conversation sizes in Tau2-Bench

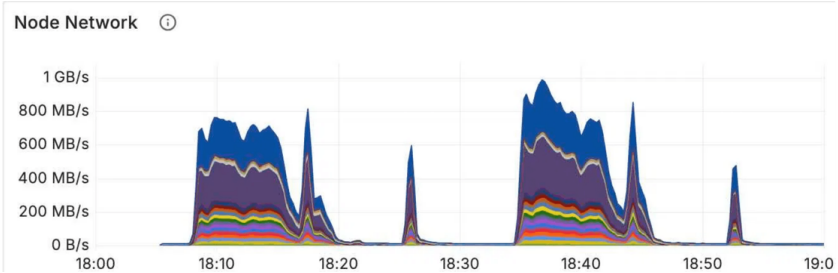



Figure 10: Total Node Network with (left) and without (right) Message Offloading

# Conclusion

## Efficient

Achieves **2–15× higher token throughput** than specialized baselines via peer-to-peer asynchronous scheduling and independent actor scaling.

## Flexible

Supports diverse multi-agent workflows (Coral, NaturalReasoning, Tau2-Bench) in a single system, configurable via Hydra and LangGraph.

## Open Source

Built on robust open-source foundations (Ray, vLLM, SGLang, Apptainer) and **fully open-sourced** to the research community.

# Limitations & Future Work

## Limitations

- **Serializable State:** Assumes per-task orchestrator state is serializable and can be passed between agents.
- **Shared Mutable State:** Less suitable when steps must read/write very large mutable shared state, where data movement or synchronization can dominate costs.
- **Ray Dependency:** Depends on the Ray actor runtime, inheriting its operational constraints and failure semantics.

## Future Work

- **Reusable Patterns:** Provide a library of reusable orchestrator patterns and end-to-end examples to minimize custom code.
- **Multi-modal Data:** Explore multi-modal data generation.
- **Continuous Synthesis:** Explore on-policy continuous data synthesis.