APWA: A DISTRIBUTED ARCHITECTURE FOR PARALLELIZABLE AGENTIC WORKFLOWS

Anonymous authors
Paper under double-blind review

ABSTRACT

Autonomous multi-agent systems based on large language models (LLMs) have demonstrated remarkable abilities in independently solving complex tasks in a wide breadth of application domains. However, these systems hit critical reasoning, coordination, and computational scaling bottlenecks as the size and complexity of their tasks grow. These limitations hinder multi-agent systems from achieving high-throughput processing for highly parallelizable tasks, despite the availability of parallel computing and reasoning primitives in the underlying LLMs. We introduce the *Agent-Parallel Workload Architecture* (APWA), a distributed multi-agent system architecture designed for the efficient processing of heavily parallelizable agentic workloads. APWA facilitates parallel execution by decomposing workflows into non-interfering subproblems that can be processed using independent resources without cross-communication. It supports heterogeneous data and parallel processing patterns, and it accommodates tasks from a wide breadth of domains. In our evaluation, we demonstrate that APWA can dynamically decompose complex queries into parallelizable workflows and scales on larger tasks in settings where prior systems fail completely. We show experimentally that APWA scales to over 4,100 agent invocations per second on a 30-node cluster, performs real tasks with over 10,000 independent subtasks, and is the only fully-automated system to complete every task at every input scale across 5 task families.

1 INTRODUCTION

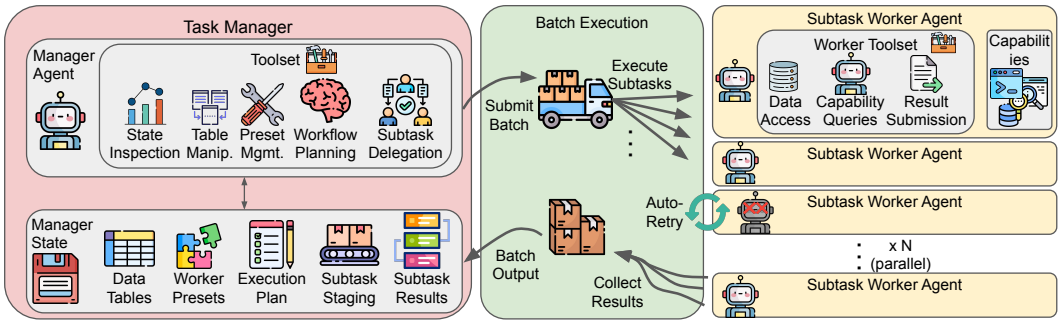


Figure 1: Overview of APWA. APWA dynamically decomposes tasks into parallelizable workflows leveraging agent workers and executes them in a distributed environment.

Autonomous LLM agents operationalize the planning, reasoning, and problem-solving capabilities of modern LLMs by situating the LLM at the center of a broader software system that interacts with an environment (Yao et al., 2023; Schick et al., 2023; Shinn et al., 2023). LLM agents have been successfully applied to domains spanning software development (Jimenez et al., 2024; Yang et al., 2024; Wang et al., 2025c), cybersecurity (Lee et al., 2025; Deng et al., 2024; Meng et al., 2024), web browsing (Deng et al., 2023; Zheng et al., 2024; Qi et al., 2025), healthcare (Kim et al., 2024b), finance (Yu et al., 2024), and scientific research (Boiko et al., 2023; M. Bran et al., 2024; Trirat et al., 2025; Chan et al., 2025; Yang et al., 2023).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

While LLM agents have enabled new applications, their LLM backbone imposes certain technical constraints. LLMs have absorbed knowledge from diverse domains, and are thus *knowledge generalists* (Brown et al., 2020; Hendrycks et al., 2021), but they may struggle to reliably modulate between multiple functional roles within a single execution trajectory (Huang et al., 2024; Gupta et al., 2024). Moreover, as tasks become larger and more complex, reasoning and output quality degrade substantially (Liu et al., 2024; Dziri et al., 2023; Hsieh et al., 2024), collapsing entirely once the size of the data exceeds the context window.

To scale agentic applications to a broader class of problems, research has explored *multi-agent systems* which coordinate several LLM agents via message passing to cohesively solve a single complex task (Hong et al., 2024; Liang et al., 2024; Qian et al., 2024; 2025; Chen et al., 2024b; Fourney et al., 2024; Wang et al., 2025b; Hu et al., 2025). Such architectures let agents assume specialized roles (Li et al., 2023; Qian et al., 2024; Hong et al., 2024; Wang et al., 2025b; Hu et al., 2025) and partition work, memory, and communication (Qian et al., 2025), yielding higher-quality outputs (Li et al., 2023; Hong et al., 2024; Wang et al., 2025a), coherence over longer time scales (Hong et al., 2024; Qian et al., 2025), and more efficient context management (Qian et al., 2025; Fourney et al., 2024).

Existing multi-agent systems nonetheless work poorly when the data exceeds a single agent's context, memory, or permanent storage, or when a task comprises a massive number of non-overlapping subproblems each requiring deeper exploration. They offer no automated task decomposition over distributed infrastructure, no scalable coordination among large agent teams, and no support for dynamic, data-dependent, heterogeneous processing. They constrain individual agent behavior by routing through global coordination and are typically built for a particular narrow application.

To better understand the problem we aim to solve, consider the paradigm shift introduced by systems designed for highly parallelizable workloads like MapReduce (Dean & Ghemawat, 2004) and Apache Spark (Zaharia et al., 2012). By providing intuitive and expressive abstractions for programmers, together with efficient, robust implementations of common data processing patterns, these systems greatly simplified the development of distributed applications.

We ask how to realize this paradigm in the agentic setting in a system that discovers and enacts a parallel execution path itself without relying on a human-authored decomposition, on tasks too large for current agent systems to complete reliably.

Such a task is not straightforward. Prior work on LLM-guided parallel workflows is limited (Kim et al., 2024a; Qiao et al., 2025; Niu et al., 2025) and does not address large-scale data processing. More fundamentally, existing agent frameworks lack abstractions for reasoning over large, distributed datasets. Unlike traditional systems, where metadata is assumed to fit in working memory, LLM-based systems must contend with settings where even metadata is too large (Section 2.2).

In this paper, we introduce the Agent-Parallel Workload Architecture (APWA), a distributed multi-agent system designed specifically for processing parallelizable workflows (Figure 1). At its core lie novel programming abstractions suited for LLM agents, allowing them to reason about large-scale data resources, decompose workflows into non-interfering subproblems, and issue and inspect distributed data-parallel execution flows. We give a scalable implementation of APWA using Ray (Moritz et al., 2018) and demonstrate that it accommodates diverse parallelizable tasks and scales to large tasks where prior systems fail completely.

Our contributions are (1) a **diagnosis** of why existing multi-agent systems fail on data- and orchestration-heavy tasks, tracing failures to inadequate agent-facing abstractions for large data, (2) new **abstractions** that resolve the tension, (3) a scalable, parallel multi-agent architecture built on top of them, and (4) an **evaluation** demonstrating that APWA scales to over 4,100 agent invocations per second on a 30-node cluster, and performs real tasks with over 10,000 independent subtasks.

2 Background and Problem Statement

2.1 Background

Several emerging architectures for multi-agent systems allow practitioners to express flexible multi-agent teams, but are not designed to accommodate distributed infrastructure, automated workload partitioning, and massively parallelized execution. More detailed discussion is given in Appendix A.

Multi-agent orchestration AutoGen (Wu et al., 2024) implements a team-based programming model in which developers manually define and instantiate agents within a common message sharing fabric. Magentic-One (Fourney et al., 2024), a multi-agent system implemented in AutoGen, uses an orchestrator-worker architecture for multi-agent cooperation. However, the synchronized message passing routed through an LLM-based orchestrator agent used in these architectures inhibits scalability and parallelization since the orchestrator must manage each dispatched agent manually, and only a single agent can process information and communicate results at a time.

Subagent delegation Some multi-agent system architectures consider parallel agent collaboration through a *subagent* design pattern. However, they either restrict to static parallelization patterns (Langchain, Inc.) or limit parallelization to a small scale by requiring expensive agent-modulated orchestration. For example, in MegaAgent (Wang et al., 2025b), a representative architecture in this category, administrator agents must *manually* create workers, cannot accommodate format divergence between peers at different layers of the hierarchy, and can not organize large data across the workers.

Parallelism at other layers Other works address parallelism at different layers. Some works build special-purpose applications such as document chunking for text summarization (Zhou et al., 2025; Zhang et al., 2024). Parallel tool calling (Kim et al., 2024a) parallelizes tool call execution, but does not affect agent-facing abstractions. Other works give efficient serving substrates (Lin et al., 2024; Liu et al., 2026) that optimize individual requests beneath the application.

2.2 Problem Statement

The problem we consider is to design a multi-agent system that automatically decomposes and executes, in parallel on distributed resources, tasks whose data volume or orchestration demands overwhelm existing agent systems. We motivate our solution through the following design goals:

① *Enable intelligent and automated task decomposition over a distributing computing and storage infrastructure.* The architecture should partition workflows into non-interfering subproblems processed on independent resources without cross-communication, and should facilitate these patterns without resource or reasoning bottlenecks.

② *Provide efficient and scalable coordination among large teams of agents,* through communication abstractions and state sharing including access to large data and a rich set of tools.

③ *Preserve individual agents' exploration capabilities*, by decoupling global coordination from local agent state so that each agent can explore freely in solving its subproblem.

④ *Support complex, dynamic, data-dependent, heterogeneous planning and processing.* The system should execute *data-parallel* (similar instructions over different subsets of data), *task-parallel* (different instructions over same inputs), and *replication-parallel* workflows (near-identical tasks and inputs with parallelization to search a large solution space faster).

⑤ *Be task- and workflow agnostic,* accommodating tasks from a wide breadth of domains rather than restricting to particular processing or parallelization patterns, and leveraging the expressivity of LLM agents to discover efficient parallel execution paths automatically.

Challenges We face several key challenges. First, while LLMs have made substantial progress in software engineering tasks (Yang et al., 2024; Jimenez et al., 2024; Wang et al., 2025c), they lack native reasoning capabilities for distributed infrastructure and large-scale parallelism, and the few works considering LLM-guided parallel workflow generation (Qiao et al., 2025; Niu et al., 2025) do not address truly large-scale data processing.

Second, existing agent scaffolding lacks abstractions for reasoning over large, distributed data resources. LLM agents typically inspect local execution state directly, which does not scale once data exceeds a single machine. Even *metadata-* or *digest-based* approaches degrade quickly: with thousands or even millions of data objects, simply enumerating identifiers for each of those objects overloads the LLM's capacity. This contrasts with traditional dataflow engines, where metadata fits in working memory. With LLM-based components, *even metadata becomes too large*.

Third, LLMs generate outputs orders of magnitude slower than traditional software. Performed through tool calls, subtask delegation itself is a bottleneck, which we term *inverted specification cost*.

These challenges influence our design in several ways (Section 3). ① *Metadata overflow* motivates a data representation decoupled from the number of items being operated upon. ② *Inverted specification cost* motivates a delegation abstraction whose cost is independent of data size. ③ *Decoupling global coordination from local worker state* makes the manager's view much more manageable.

3 THE APWA SYSTEM

APWA is a distributed multi-agent system designed specifically for highly parallelizable agentic workflows. At the core of APWA lies a set of abstractions that together achieve the five goals of Section 2.2: system-level abstractions supporting high-level parallelization goals (Section 3.1), and agent-level abstractions supporting automated task decomposition and execution (Section 3.2).

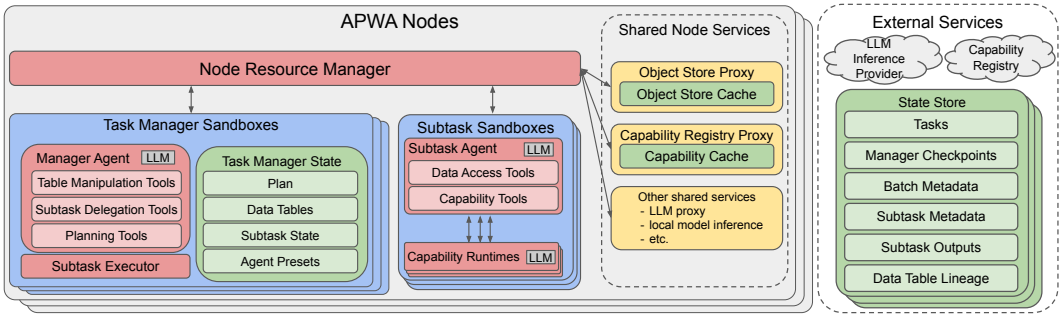


Figure 2: APWA Distributed System Architecture.

3.1 APWA SYSTEM-LEVEL ABSTRACTIONS

Like other multi-agent systems, APWA is organized around a *manager*, *workers*, and *executor*. However, our abstractions differ in targeting data-parallel and task-parallel workflows. Both the manager and worker modules plan: the manager performs meta-planning by decomposing the task into parallelizable subtasks, while each worker plans how to solve its assigned subtask. The executor performs the actual execution in a distributed cluster, hiding low-level distributed computing details.

Manager The manager is responsible for solving a single task. It dynamically partitions the task into non-interfering *subtasks* which can be dispatched and executed in parallel, supported by (1) a definition of a general parallelizable unit, (2) guided planning with examples of parallelization patterns, and (3) abstractions for passing large data to workers. Subtasks specify contracts for a unit of work and the requested worker configuration. The manager is the only component with a global view of a task's execution state and the only one performing high-level task reasoning.

Worker A worker solves a single subtask. A task may involve many workers, ranging in complexity from simple LLM processors to multi-agent subsystems with specialized roles. Workers also receive support for parallelization, including guidance on their role in the system, access to a wide set of agents and tools, and abstractions for creating and sharing data. A worker's view of system state is strictly *local*, comprising the subtask inputs and contents of their local execution space.

Executor The executor collects execution requests from the manager, runs them, and reports the results. It differs from other systems in its focus on leveraging distributed resources of a cluster. It places subtasks on cluster nodes and handles timeouts, automatic retries for transient errors, and principled collection of results. By abstracting away execution details like worker placement, resource allocation, failure recovery, and retry logic, the executor frees the manager to reason purely about the logical structure and semantics of subtasks instead of low-level distributed infrastructure.

3.2 AGENT-FACING ABSTRACTIONS

We now introduce abstractions which enable the manager and workers to reason over and manipulate execution state and distributed data resources, and define and issue high volume of parallel subtasks.

Planning for parallelization We design a planning component tailored for parallelizable workflows. On starting a task the manager explores local state to understand the task and input data, then generates a structured plan object containing a list of procedural steps that it will maintain for the task's life cycle. The plan's fields describe how to decompose the task into subtasks and how data and intermediate results should be partitioned or reorganized. The manager may also create an *output contract* defining structures that must exist before the task can be considered complete. This contract is checked deterministically against true system state as a feedback signal and prevents premature termination.

Subtask delegation This abstraction lets the manager structurally describe subtask without manually populating its details from data. It thus allows expressing an entire *batch* of subtasks at once. We introduce *subtask templates*, which describe a parameterized unit of work containing fields for agent configuration, task to be completed, and references to data resources to process. Template input parameters may contain placeholders that expand automatically when paired with a data resource, decoupling the logical content of a subtask from the scale of the data it acts upon.

Data tables In a distributed execution setting, some tasks might involve large data objects that collectively exceed the LLM's context window or even the memory constraints of the machine. We argue a good data abstraction should satisfy the following principles: factor out common structure, organize items into manipulable collections, and map cleanly onto execution units.

We introduce a *data table* abstraction satisfying these desiderata. A data table is a logical, read-only, finite sequence of records conforming to a common *schema*. Data tables are quite similar to related notions in data analysis libraries and databases. It provides a constant-size metadata representation, groups collections as rows under a table algebra, and coupled directly to subtask templates, which expand over a table's rows. Defining tables algebraically enables fault tolerance via *lineage*.

The manager interacts with tables through a dedicated tool suite to inspect, query, analyze, manipulate, and construct tables dynamically. This lets it gather information about intermediate execution state and prepare data for future processing. We define two categories of table-focused tools: *table analytics* tools allow the task manager agent to query information about data stored in tables, and *table manipulation* tools allow the task manager to construct new tables from existing ones. A full set of table tools and their descriptions are given in Appendix B.1.2.

Dynamic agent capabilities and presets Our architecture makes no assumptions about which agent-powered software capabilities are required by a task. Moreover, those capabilities may be known only at runtime, perhaps as part of the task specification itself. The system should therefore allow specialized agents to be defined, constructed, and executed at runtime, which we realize with an *agent capability and preset* abstraction. This abstraction lets the manager identify and use whatever specialized tools a task requires.

Capabilities are discovered through an externally hosted *Capability Registry* which exposes diverse software functionalities for use during subtask execution. These capabilities enable dynamic, composable functionalities for workers. To define agents using capabilities, the manager creates an *agent preset* defining a system prompt and a set of capabilities for the worker agent to use. When delegating subtasks, the manager agent specifies which preset to use.

Our capability mechanism differs from tool-discovery protocols like MCP (Model Context Protocol a Series of LF Projects, LLC.), which implement tool logic server-side. Our registry does not itself execute tool logic, instead hosting implementations of the tools which run locally within each worker's execution space.

3.3 APWA IMPLEMENTATION

Figure 2 depicts the architecture of our system using the abstractions described above. At runtime, APWA processes a series of *rounds* cycling between two phases: a high-level reasoning and delegation phase performed by the manager and a low-level subtask execution phase performed by workers. Both the manager and the worker are implemented as agents running in sandboxed environments.

Manager environment The manager oversees a single task to completion by planning processing steps, observing and manipulating local data representations, and dispatching units of work for remote

execution. It tracks progress by maintaining a highly-compact metadata representation of global system state, a collection of workers, an actively-maintained execution plan, a detailed output format specification, and a log of results from previous rounds.

The manager's state is persisted in an external state store recording task metadata, manager checkpoints, issued subtask batches, subtask outputs, and synthesized results. The store is the interface through which the manager interacts with system state such as outputs from completed subtasks.

Worker environment The worker's execution environment is equipped with whatever auxiliary runtime capabilities are needed to complete its subtask, such as virtual machine resources (local file system, terminal capabilities, etc.) or network-accessible services. The worker has tools for reading and writing data to a shared network-accessible data resource.

We observe that frequently agent execution patterns reduce to a simple LLM invocation. For example, simple data processing steps like summarization or structured information extraction involve only a single LLM generation step. In these cases, we can improve subtask throughput by a substantial factor (10-100$\times$) by dynamically routing the subtask execution to a lightweight execution space. This routing is determined dynamically by the manager when it emits a subtask based on its own assessment of the complexity of the subtask.

Scalable execution fabric We use the Ray distributed computing framework (Moritz et al., 2018) to provide APWA's distributed computing fabric. Ray allows fault-tolerant distributed subtask execution on a cluster of machines. Failed subtasks are automatically retried up to a fixed number of times before returning a logical failure to the task manager, allowing it reason about the *logical* content of its subtasks rather than low-level distributed execution behaviors.

4 Evaluation

4.1 Setup

Benchmarks We evaluate APWA on five task families: hierarchical summarization, PII redaction, data format conversion, multilingual translation, and web report synthesis. The `SummaryBench` task is to generate summaries at each level of a hierarchically organized text corpus. We build this task from three texts available on Project Gutenberg (Project Gutenberg): *Romeo and Juliet* (R&J, 2 tiers, 166 kB), *The Dynasts* (3 tiers, 882 kB), and *The History of the Decline and Fall of the Roman Empire* (*Roman*, 4 tiers, 10.2 MB) (Shakespeare, 1998; Hardy, 2005; Gibbon, 2008). The `PII-300k` task is to redact 27 categories of sensitive information from a table of records (Ai4Privacy, 2024). The `SchemaBench` task is to generate schema-conformant JSON extracted from four suites of heterogeneous documents (LaTeX, XML, CSV, HTML) (Bai et al., 2024). The `TranslationBench` task, based on the FLORES+ dataset (Costa-jussà et al., 2024), is to translate every sentence in a table to every language variety in a second table. Finally, the `ReportBench` task is to generate reports on a list of 10, 20, or 100 topics, which requires web-browsing agents as workers. Appendix C describes each benchmark in full detail.

Metrics We define a *structural* and *semantic* score for each task. The structural score checks that the correct outputs exist with the correct format. The semantic score measures output correctness. We summarize the different semantic scores used for each task in Table 1 with details in Appendix C. We define trial *failure* to mean the method returns no output or its harness reports a failure. Note a low failure rate does not imply useful output. We also report wall clock runtime and USD API cost.

Table 1: Tasks and semantic scores.

Task	Semantic score
`PII-300k`	Macro-averaged F_1
`SchemaBench`	Word-level F_1
`SummaryBench`	ROUGE-1 F_1
`TranslationBench`	chrF++
`ReportBench`	LLM judge

Baselines We consider four baseline methods: *Direct*, *Magentic-One* (Fourney et al., 2024), *MegaAgent* (Wang et al., 2025b), and *LangChain subagents* (Langchain, Inc.). Direct sends the task and all input data directly to a single LLM invocation with structured output. Magentic-One is a multi-agent system with built-in file system, web browser, and coding agents. It executes workers serially with all inter-agent coordination mediated by a central planning and routing module. MegaAgent is a parallel multi-agent architecture with hierarchical agent coordination. A top-level *boss* agent partitions work

Table 2: Structural and semantic scores for `SummaryBench` and `PII-300k`. All settings use GPT-5.4 mini, except MegaAgent using GPT-4.1 mini. Scores computed only for trials that produced output. Str. = structural score, Sem. = semantic score.

	R&J		Dynasts		Roman		PII-64		PII-512		PII-4096	
Method	**Str. (↑)**	**Sem. (↑)**	**Str. (↑)**	**Sem. (↑)**	**Str. (↑)**	**Sem. (↑)**	**Str. (↑)**	**Sem. (↑)**	**Str. (↑)**	**Sem. (↑)**	**Str. (↑)**	**Sem. (↑)**
Direct	1.00	0.43	0.98	0.21	⊥	⊥	1.00	0.78	0.75	0.16	⊥	⊥
Magentic-One	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	1.00	0.18
MegaAgent	0.14	0.04	0.21	0.02	0.16	0.02	0.38	0.00	0.00	0.00	0.25	0.00
LangChain (flat)	1.00	0.46	0.90	0.26	0.12	0.03	0.83	0.04	0.75	0.01	0.50	0.00
LangChain (recursive)	0.99	0.41	0.91	0.20	0.22	0.03	0.83	0.04	0.75	0.00	0.75	0.00
APWA	0.95	0.42	0.97	0.45	0.91	0.23	1.00	0.76	1.00	0.77	0.90	0.54

Table 3: Failure rates and wall-clock runtimes for `SummaryBench` and `PII-300k`. All settings use GPT-5.4 mini, except MegaAgent using GPT-4.1 mini. Wall-clock runtimes reflected only for trials that produced output. FR = failure rate, WC = wall-clock time (seconds).

	R&J		Dynasts		Roman		PII-64		PII-512		PII-4096	
Method	**FR (↓)**	**WC (s) (↓)**	**FR (↓)**	**WC (s) (↓)**	**FR (↓)**	**WC (s) (↓)**	**FR (↓)**	**WC (s) (↓)**	**FR (↓)**	**WC (s) (↓)**	**FR (↓)**	**WC (s) (↓)**
Direct	0%	19	60%	76	100%	⊥	0%	37	0%	41	100%	⊥
Magentic-One	100%	⊥	100%	⊥	100%	⊥	100%	⊥	100%	⊥	80%	91
MegaAgent	60%	472	60%	248	40%	579	60%	390	80%	25	70%	372
LangChain (flat)	0%	54	20%	145	20%	283	40%	136	40%	80	20%	23
LangChain (recursive)	0%	33	40%	111	40%	56	40%	103	20%	136	20%	30
APWA	0%	96	0%	142	0%	272	0%	35	0%	67	0%	221

by spawning subagent teams, which recursively repeat the process until work is complete or a limit is reached. [1] Finally, we implement code agents with subagent delegation capabilities with the LangChain subagents package. We evaluate both *flat* and *recursive* delegation separately.

We also implement and evaluate hand-authored decompositions of each task under sequential execution and parallel execution with Ray and Python's `asyncio` module.

System configuration Main experiments run on one Ubuntu 22.04.5 machine with a 16-Core AMD Ryzen Threadripper PRO 5955WX processor, 252 GB memory, and two NVIDIA GeForce RTX 4090 GPUs. We pair GPT-5.4, GPT-5.4 mini, or GPT-5.4 nano (OpenAI, 2025a;b) as manager and worker LLMs, denoted *manager*×*worker*.

4.2 Main results

Baselines cannot process large inputs Table 2 and Table 3 contain results for `SummaryBench` and `PII-300k`. Direct performs well only for small inputs (R&J, PII-64), where semantic score is near the hand-authored baseline.

Magentic-One is unable to reliably complete tasks, with only a single trial producing output for PII-4096. We identify *orchestration failure* and *context explosion* as failure modes. Orchestration failure occurred when the orchestrator incorrectly routed requests between its Coder and CodeExecutor agents, causing both subagents to fail to complete any useful work. Context explosion occurred on largest inputs: for the *Roman* corpus, the FileSurfer agent attempted to page through over 2,500 input documents and consumed over 6.5M input tokens until a timeout at 30 minutes.

MegaAgent and LangChain fail differently from Magentic-One. Both systems use their harness to delegate work to subagents. However, the delegation is ineffective. In one `SummaryBench` trial, MegaAgent delegates work between `SceneSummarizer`, `ActSummarizer`, and `FullSummarizer` agents. `SceneSummarizer` fails to process scene files quickly and the task is terminated. The LangChain supervisor also fails to scale delegation with input size. On PII-4096, it requests between 1 and 8 workers per trial, often passing individual files to workers. The

[1] MegaAgent's implementation uses an tool-calling API not supported by more recent models. We choose GPT-4.1 mini as a comparable surrogate.

Table 4: Comparison of APWA configurations on `SummaryBench`. Str. = structural score, Sem. = semantic score, WT = wall-clock time (sec), Cost = total cost.

	R&J				Dynasts				Roman			
Method	**Str. (↑)**	**Sem. (↑)**	**WT (s) (↓)**	**Cost (↓)**	**Str. (↑)**	**Sem. (↑)**	**WT (s) (↓)**	**Cost (↓)**	**Str. (↑)**	**Sem. (↑)**	**WT (s) (↓)**	**Cost (↓)**
APWA (`mini×mini`)	0.95	0.42	96	$0.244	0.97	0.45	142	$0.684	0.91	0.23	272	$5.32
APWA (`mini×nano`)	0.95	0.43	85	$0.17	0.87	0.32	142	$0.377	0.91	0.30	281	$1.98
APWA (`5.4×mini`)	1.00	0.53	168	$0.628	1.00	0.45	235	$1.16	0.98	0.37	372	$6.57
APWA (`5.4×nano`)	0.94	0.44	174	$0.582	0.92	0.42	234	$0.919	0.87	0.40	390	$3.09

orchestrator then writes a final answer containing at most 15 of 4096 documents. In other trials, the orchestrator fails to aggregate outputs produced by workers.

APWA processes large inputs APWA processes large datasets efficiently while keeping output quality near the hand-authored baseline. The mean structural score on all tasks is at least 0.9, meaning APWA produces outputs with the correct schemas and sizes. The semantic scores of 0.42, 0.45, and 0.23 on `SummaryBench`, and 0.76, 0.77, and 0.54 on `PII-300k` are near the hand-authored decompositions. On small inputs it matches the quality of successful baselines: on R&J, Direct, LangChain (flat and recursive), and APWA all attain semantic scores between 0.41-0.46.

System configurations Table 4 evaluates APWA with different combinations of manager and worker LLM. Both GPT-5.4 and GPT-5.4 mini are able to use APWA's abstractions effectively. Larger GPT-5.4 as the manager yields higher structural and semantic scores at mild cost increase (1.23–1.56×) compared with GPT-5.4 mini.

Overhead vs. manual decomposition We measure APWA's runtime against manual hand-authored parallel decompositions. These decompositions serve as practical "best case scenarios" for APWA's runtime and output quality, since APWA must perform the task decomposition itself in addition to the execution. Despite the size of input data increasing by nearly two orders of magnitude (163 kB on *Romeo and Juliet* → 10 203 kB on *Roman*), APWA's wall-clock runtime increases only moderately (96 s → 272 s) due to massive parallelization, running over 2.5k agents concurrently. APWA is roughly 7.8x slower than the Ray manual decomposition for *Romeo and Juliet*, shrinking to 3.2x overhead for *Decline and Fall*. Thus, as fan-out size grows, the orchestration overhead becomes a smaller fraction of total task runtime. Under sequential execution, the baseline is faster than APWA for *Romeo and Juliet*, slower for *The Dynasts*, and times out after 2 h on *Roman*.

APWA errors While APWA outperforms baselines, it is not perfect. We observe in some trials that APWA does not pass inputs correctly to workers, resulting in empty results, while in others APWA performs delegation correctly but does not derive the output column correctly (and so scores a 0 despite nearly solving the task). These failure cases are minor and can be overcome with a more mature harness (e.g., extra data validation and better system prompts) and more powerful LLMs.

4.3 Additional Experiments

Scalability study We measure the scalability of the execution layer on Ray clusters of 1–30 m6i.4xlarge AWS EC2 instances with representative deterministic subtasks for both execution modes. We measure throughput exceeding 4,100 subtasks per second in the LLM-only mode and 30 subtasks per second in the full environment (Figure 3), with weak scaling efficiency of 0.74 and 0.97 for the different execution modes, respectively. More details are given in Appendix C.6.

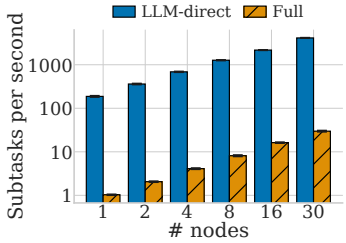


Figure 3: APWA executor throughput scaling.

Data extraction benchmark Our qualitative findings on `SchemaBench` (Appendix C.3) are similar to our findings on `SummaryBench` and `PII-300k`. On the only dataset whose size is small enough to fit within the 1M token limit for GPT-5.4 mini, CHEMISTRY, the semantic scores for APWA are comparable to the scores for querying the LLM directly. For all other datasets, the

input size exceeds the LLM context window when formatted directly. Magentic-One only manages to produce output in 4 out of 20 cases. Like in `SummaryBench` and `PII-300k`, Magentic-One pursues programmatic solutions that do not meet the complex semantic nature of the task. MegaAgent also fails to complete the structured extraction task, again due to a failure to parallelize work and instead attempt purely role-based task decomposition. Out of 20 attempted tasks, only 6 produced any output, only 3 of which received nonzero scores. Of these, the semantic score is always 0.

Full agentic workload To demonstrate the utility of APWA in orchestrating fully agentic workloads, we consider a technical report synthesis task leveraging web browsing agents. The task definition is to generate textual reports on a list of topics provided in the input. We consider three variants with a small (10), medium (20), and large (100) number of topics, respectively. We populate the capability registry with Magentic-One's web surfer agent and expose its description to the manager agent. We measure whether the correct number of (topic, report) pairs are generated and use LLM-as-a-judge to measure whether the report is factual and on-topic. In experiments, APWA successfully identifies the web surfer agent as relevant and creates an appropriate subtask template to pass input topics to workers. Full results are given in Appendix C.4.

Task-parallel workloads To demonstrate the task-parallel pattern of design goal ④, we use a multilingual translation task. We consider this task task-parallel because the input data (an English sentence) is fixed, while the instruction being executed ("translate the document into language X") varies for each subtask. We translate sentences into each of 213 FLORES+ target language varieties. The task input comprises one table of sentences and one table of target languages. Solving the task requires taking the cross product of the two tables first, then to issue subtasks over the resulting items. APWA automatically recognizes this structure without extra instruction, successfully expanding into up to $> 13,500$ subtasks and completing the task on average in just over $6\,\mathrm{min}$. We report detailed metrics in Appendix C.5.

Subtask template ablation Fixing the data abstraction, executor, and manager LLM, we remove the manager's subtask template so that it must delegate each subtask individually, following the standard subagent pattern. The ablated manager fails every `SummaryBench` corpus: tool-calling budgets prevent it from scheduling all required subtask, and combining results across rounds is error-prone, resulting in corrupted intermediate state. This experiment demonstrates the importance of APWA's efficient delegation mechanism.

5 Conclusion

In this work, we propose APWA, a highly scalable multi-agent execution architecture for parallelizable agentic workflows. APWA introduces efficient abstractions for LLM-based agents to reason about and act upon high volumes of distributed data and computing resources and to specify and issue large-scale parallel execution of agents. We demonstrate that APWA is capable of automatically and efficiently solving large-scale distributed workflows using thousands of agents, in contrast to existing methods which cannot effectively leverage parallelism and fail on larger complex tasks.

Limitations APWA is designed for highly parallelizable workloads. Its effectiveness depends heavily on the parallelizable nature of the task. We observe in experiments (Section 4) that APWA is preferable to traditional systems once the number of subtasks reaches 100s. For smaller tasks, or tasks with heavy inter-subtask dependencies, APWA is unlikely to provide advantages over typical architectures.

APWA deliberately prohibits communication channels between workers. This greatly simplifies the semantics of batch execution: every subtask is independent; failures can be retried; and data does not need to be transmitted between workers. Within a worker, communication is still permitted: a worker can itself be a small multi-agent team. Still, some applications might benefit from large-scale inter-agent coordination, such as software development of a large code base. Future work may explore principled designs for such use cases.

AI USE STATEMENT

In this work, we used generative AI tools to implement methods, format datasets, search for information, and discover relevant literature. We have not used generative AI tools to generate synthetic data sets, propose or refine hypotheses, design or provide feedback on research methodology or experiments, assist with translation, support qualitative and thematic data analysis, or interpret results. Developing theoretical models or conceptual frameworks, formulating mathematical claims, providing critical ingredients for proving mathematical claims, and assisting in the writing of proofs are not applicable to this work. We have reviewed all AI-assisted work. All LLM-generated code was verified manually. Activities involving discovery of prior work were conducted with multiple methods including LLM-guided deep research, academic search engines, and manual citation. All bibliographic metadata was collected manually. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We make the code available at `https://anonymous.4open.science/r/autogen-playground-6EFB`. Appendix B describes our proposed system architecture in detail. We use public datasets in our evaluation, all of which are explained in Appendix C.

REFERENCES

Ai4Privacy. PII masking 300k dataset. `https://huggingface.co/datasets/ai4privacy/pii-masking-300k`, 2024. URL `https://huggingface.co/datasets/ai4privacy/pii-masking-300k`.

Apache Software Foundation. Hadoop. `https://hadoop.apache.org/`.

Fan Bai, Junmo Kang, Gabriel Stanovsky, Dayne Freitag, Mark Dredze, and Alan Ritter. Schema-driven information extraction from heterogeneous tables. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 10252–10273, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-emnlp.600. URL `https://aclanthology.org/2024.findings-emnlp.600/`.

Daniil A. Boiko, Robert MacKnight, Ben Kline, and Gabe Gomes. Autonomous chemical research with large language models. *Nature*, 624(7992):570–578, December 2023. ISSN 1476-4687. doi: 10.1038/s41586-023-06792-0. URL `https://doi.org/10.1038/s41586-023-06792-0`.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020. URL `https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf`.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Aleksander Madry, and Lilian Weng. MLE-bench: Evaluating machine learning agents on machine learning engineering. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 50466–50494, 2025. URL `https://proceedings.iclr.cc/paper_files/paper/2025/file/7e3767db483c942b883eb4f8cfb74e31-Paper-Conference.pdf`.

Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje Karlsson, Jie Fu, and Yemin Shi. AutoAgents: A framework for automatic agent generation. In Kate Larson (ed.),

Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24, pp. 22–30. International Joint Conferences on Artificial Intelligence Organization, 8 2024a. doi: 10.24963/ijcai.2024/3. URL https://doi.org/10.24963/ijcai.2024/3. Main Track.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. AgentVerse: Facilitating multi-agent collaboration and exploring emergent behaviors. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 20094–20136, 2024b. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/578e65cdee35d00c708d4c64bce32971-Paper-Conference.pdf.

Marta R. Costa-jussà, James Cross, Onur Çelebi, Maha Elbayad, Kenneth Heafield, Kevin Heffernan, Elahe Kalbassi, Janice Lam, Daniel Licht, Jean Maillard, Anna Sun, Skyler Wang, Guillaume Wenzek, Al Youngblood, Bapi Akula, Loic Barrault, Gabriel Mejia Gonzalez, Prangthip Hansanti, John Hoffman, Semarley Jarrett, Kaushik Ram Sadagopan, Dirk Rowe, Shannon Spruit, Chau Tran, Pierre Andrews, Necip Fazil Ayan, Shruti Bhosale, Sergey Edunov, Angela Fan, Cynthia Gao, Vedanuj Goswami, Francisco Guzmán, Philipp Koehn, Alexandre Mourachko, Christophe Ropers, Safiyyah Saleem, Holger Schwenk, Jeff Wang, and N. L. L. B. Team. Scaling neural machine translation to 200 languages. *Nature*, 630(8018):841–846, June 2024. ISSN 1476-4687. doi: 10.1038/s41586-024-07335-x. URL https://doi.org/10.1038/s41586-024-07335-x.

CrewAI, Inc. CrewAI. https://www.crewai.com/.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th Symposium on Operating Systems Design and Implementation (OSDI 04)*, pp. 137–150, San Francisco, CA, December 2004. USENIX Association. URL https://www.usenix.org/conference/osdi-04/mapreduce-simplified-data-processing-large-clusters.

Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. PentestGPT: Evaluating and harnessing large language models for automated penetration testing. In *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 847–864, Philadelphia, PA, August 2024. USENIX Association. ISBN 978-1-939133-44-1. URL https://www.usenix.org/conference/usenixsecurity24/presentation/deng.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2Web: Towards a generalist agent for the web. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 28091–28114. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/5950bf290a1570ea401bf98882128160-Paper-Datasets_and_Benchmarks.pdf.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt. *ACM Trans. Softw. Eng. Methodol.*, 33(7), September 2024. ISSN 1049-331X. doi: 10.1145/3672459. URL https://doi.org/10.1145/3672459.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 11733–11763. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/du24e.html.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang (Lorraine) Li, Liwei Jiang, Bill Yuchen Lin, Sean Welleck, Peter West, Chandra Bhagavatula, Ronan Le Bras, Jena Hwang, Soumya Sanyal, Xiang Ren, Allyson Ettinger, Zaid Harchaoui, and Yejin Choi. Faith and fate: Limits of transformers on compositionality. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 70293–70332. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/p

aper/2023/file/deb3c28192f979302c157cb653c15e90-Paper-Conference.pdf.

Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Erkang (Eric) Zhu, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, Peter Chang, Ricky Loynd, Robert West, Victor Dibia, Ahmed Awadallah, Ece Kamar, Rafah Hosn, and Saleema Amershi. Magentic-One: A generalist multi-agent system for solving complex tasks. Technical Report MSR-TR-2024-47, Microsoft, November 2024. URL https://www.microsoft.com/en-us/research/publication/magentic-one-a-generalist-multi-agent-system-for-solving-complex-tasks/.

Edward Gibbon. *The History of the Decline and Fall of the Roman Empire*. Project Gutenberg, Urbana, Illinois, 2008. URL https://www.gutenberg.org/ebooks/25717. Accessed: 2026-04-27.

Akash Gupta, Ivaxi Sheth, Vyas Raina, Mark Gales, and Mario Fritz. LLM task interference: An initial study on the impact of task-switch in conversational history. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 14633–14652, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.811. URL https://aclanthology.org/2024.emnlp-main.811/.

Thomas Hardy. *The Dynasts: An Epic-Drama of the War with Napoleon*. Project Gutenberg, Urbana, Illinois, 2005. URL https://www.gutenberg.org/ebooks/4043. Accessed: 2026-04-27.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=d7KBjmI3GmQ.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VtmBAGCN7o.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, and Boris Ginsburg. RULER: What's the real context size of your long-context language models? In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=kIoBbc76Sy.

Mengkang Hu, Yuhang Zhou, Wendong Fan, Yuzhou Nie, Ziyu Ye, Bowei Xia, Tao Sun, Zhaoxuan Jin, Yingru Li, Zeyu Zhang, Yifeng Wang, Qianshuo Ye, Bernard Ghanem, Ping Luo, and Guohao Li. Owl: Optimized workforce learning for general multi-agent assistance in real-world task automation. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, pp. 50859–50906. Curran Associates, Inc., 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/48dcc43a534c5b582f9d0fdb778e9b84-Paper-Conference.pdf.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 32808–32824, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/8b4add8b0aa8749d80a34ca5d941c355-Paper-Conference.pdf.

Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pp. 59–72, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936363. doi: 10.1145/1272996.1273005. URL https://doi.org/10.1145/1272996.1273005.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world github issues? In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/edac78c3e300629acfe6cbe9ca88fb84-Paper-Conference.pdf.

Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. An LLM compiler for parallel function calling. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 24370–24391. PMLR, 21–27 Jul 2024a. URL https://proceedings.mlr.press/v235/kim24y.html.

Yubin Kim, Chanwoo Park, Hyewon Jeong, Yik Siu Chan, Xuhai Xu, Daniel McDuff, Hyeonhoon Lee, Marzyeh Ghassemi, Cynthia Breazeal, and Hae Won Park. Mdagents: An adaptive collaboration of llms for medical decision-making. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 79410–79452. Curran Associates, Inc., 2024b. doi: 10.52202/079017-2522. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/90d1fc07f46e31387978b88e7e057a31-Paper-Conference.pdf.

Langchain, Inc. LangChain. https://github.com/langchain-ai/langchain/.

Hwiwon Lee, Ziqi Zhang, Hanxiao Lu, and Lingming Zhang. SEC-bench: Automated benchmarking of llm agents on real-world software security tasks. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, pp. 116342–116378. Curran Associates, Inc., 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/a9168f1c54e5147027f1e8cf83e1a775-Paper-Conference.pdf.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: Communicative agents for "mind" exploration of large language model society. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 51991–52008. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/a3621ee907def47c1b952ade25c67698-Paper-Conference.pdf.

Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Shuming Shi, and Zhaopeng Tu. Encouraging divergent thinking in large language models through multi-agent debate. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 17889–17904, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.992. URL https://aclanthology.org/2024.emnlp-main.992/.

Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient serving of LLM-based applications with semantic variable. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 929–945, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/lin-chaofan.

Jun Liu, Peilin Liu, Ruicheng Zhang, Zhang Senlei, Yanbo Chen, Ziao Wang, Jinyun Yang, mingqi wang, Shuhao Zhang, Xiaofei Liao, and Hai Jin. SAGE: A dataflow-native framework for modular, controllable, and transparent LLM-augmented reasoning. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=TXcFJdT7at.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024. doi: 10.1162/tacl_a_00638. URL https://aclanthology.org/2024.tacl-1.9/.

Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E. Gonzalez, and Ion Stoica. Autellix: An efficient serving engine for llm agents as general programs, 2025. URL https://arxiv.org/abs/2502.13965.

Andres M. Bran, Sam Cox, Oliver Schilter, Carlo Baldassari, Andrew D. White, and Philippe Schwaller. Augmenting large language models with chemistry tools. *Nature Machine Intelligence*, 6(5):525–535, May 2024. ISSN 2522-5839. doi: 10.1038/s42256-024-00832-8. URL https://doi.org/10.1038/s42256-024-00832-8.

Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pp. 135–146, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300322. doi: 10.1145/1807167.1807184. URL https://doi.org/10.1145/1807167.1807184.

Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. Large language model guided protocol fuzzing. In *Proceedings 2024 Network and Distributed System Security Symposium*, NDSS 2024. Internet Society, 2024. doi: 10.14722/ndss.2024.24556. URL http://dx.doi.org/10.14722/ndss.2024.24556.

Microsoft Corporation. Microsoft agent framework. https://github.com/microsoft/agent-framework/.

Model Context Protocol a Series of LF Projects, LLC. Model context protocol (MCP). https://github.com/modelcontextprotocol/modelcontextprotocol.

Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 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.

Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. CIEL: A universal execution engine for distributed Data-Flow computing. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, March 2011. USENIX Association. URL https://www.usenix.org/conference/nsdi11/ciel-universal-execution-engine-distributed-data-flow-computing.

Boye Niu, Yiliao Song, Kai Lian, Yifan Shen, Yu Yao, Kun Zhang, and Tongliang Liu. Flow: Modularized agentic workflow automation. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 74949–74977, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/ba84da6921f3040b74ee163aa7451f53-Paper-Conference.pdf.

OpenAI. Introducing GPT-5.4. https://openai.com/index/introducing-gpt-5-4/, 2025a. Accessed: 2026-04-27.

OpenAI. Introducing GPT-5.4 mini and nano. https://openai.com/index/introducing-gpt-5-4-mini-and-nano/, 2025b. Accessed: 2026-04-27.

OpenAI. Openai agents sdk. https://github.com/openai/openai-agents-python, 2026. Accessed: 2026-05-05.

Liana Patel, Siddharth Jha, Melissa Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. Semantic operators: A declarative model for rich, AI-based data processing, 2025. URL https://arxiv.org/abs/2407.11418.

Maja Popović. chrF++: words helping character n-grams. In Ondřej Bojar, Christian Buck, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Matthias Huck, Antonio Jimeno Yepes, Philipp Koehn, and Julia Kreutzer (eds.), *Proceedings of the Second Conference on Machine*

Translation, pp. 612–618, Copenhagen, Denmark, September 2017. Association for Computational Linguistics. doi: 10.18653/v1/W17-4770. URL https://aclanthology.org/W17-4770/.

Matt Post. A call for clarity in reporting BLEU scores. In Ondřej Bojar, Rajen Chatterjee, Christian Federmann, Mark Fishel, Yvette Graham, Barry Haddow, Matthias Huck, Antonio Jimeno Yepes, Philipp Koehn, Christof Monz, Matteo Negri, Aurélie Névéol, Mariana Neves, Matt Post, Lucia Specia, Marco Turchi, and Karin Verspoor (eds.), *Proceedings of the Third Conference on Machine Translation: Research Papers*, pp. 186–191, Brussels, Belgium, October 2018. Association for Computational Linguistics. doi: 10.18653/v1/W18-6319. URL https://aclanthology.org/W18-6319/.

Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, October 2010. USENIX Association. URL https://www.usenix.org/conference/osdi10/piccolo-building-fast-distributed-programs-partitioned-tables.

Project Gutenberg. Project gutenberg. https://www.gutenberg.org/. URL https://www.gutenberg.org/. Accessed: 2026-04-27.

Zehan Qi, Xiao Liu, Iat Long Iong, Hanyu Lai, Xueqiao Sun, Jiadai Sun, Xinyue Yang, Yu Yang, Shuntian Yao, Wei Xu, Jie Tang, and Yuxiao Dong. Webrl: Training llm web agents via self-evolving online curriculum reinforcement learning. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 79791–79821, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/c66e1fcc9691aae706250638f36f681b-Paper-Conference.pdf.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15174–15186, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.810. URL https://aclanthology.org/2024.acl-long.810/.

Chen Qian, Zihao Xie, YiFei Wang, Wei Liu, Kunlun Zhu, Hanchen Xia, Yufan Dang, Zhuoyun Du, Weize Chen, Cheng Yang, Zhiyuan Liu, and Maosong Sun. Scaling large language model-based multi-agent collaboration. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 41488–41505, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/66a026c0d17040889b50f0dfa650e5e0-Paper-Conference.pdf.

Shuofei Qiao, Runnan Fang, Zhisong Qiu, Xiaobin Wang, Ningyu Zhang, Yong Jiang, Pengjun Xie, Fei Huang, and Huajun Chen. Benchmarking agentic workflow generation. In *Workshop on Reasoning and Planning for Large Language Models*, 2025. URL https://openreview.net/forum?id=1VU7zLtQyW.

Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In Kathryn Huff and James Bergstra (eds.), *Proceedings of the 14th Python in Science Conference*, pp. 126–132, 2015. doi: 10.25080/Majora-7b98e3ed-013.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 68539–68551. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/d842425e4bf79ba039352da0f658a906-Paper-Conference.pdf.

William Shakespeare. *Romeo and Juliet*. Project Gutenberg, Urbana, Illinois, 1998. URL https://www.gutenberg.org/ebooks/1513. Accessed: 2026-04-27.

Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G. Parameswaran, and Eugene Wu. DocETL: Agentic query rewriting and evaluation for complex document processing, 2025. URL https://arxiv.org/abs/2410.12189.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf.

Patara Trirat, Wonyong Jeong, and Sung Ju Hwang. AutoML-agent: A multi-agent LLM framework for full-pipeline AutoML. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 60099–60146. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/trirat25a.html.

Junlin Wang, Jue Wang, Ben Athiwaratkun, Ce Zhang, and James Y Zou. Mixture-of-agents enhances large language model capabilities. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 33944–33963, 2025a. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/5434be94e82c54327bb9dcaf7fca52b6-Paper-Conference.pdf.

Qian Wang, Tianyu Wang, Zhenheng Tang, Qinbin Li, Nuo Chen, Jingsheng Liang, and Bingsheng He. MegaAgent: A large-scale autonomous LLM-based multi-agent system without predefined SOPs. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 4998–5036, Vienna, Austria, July 2025b. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.259. URL https://aclanthology.org/2025.findings-acl.259/.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Daniel Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025c. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/a4b6ad6b48850c0c331d1259fc66a69c-Paper-Conference.pdf.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=BAakY1hNKS.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652. Curran Associates, Inc., 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5a7c947568c1b1328ccc5230172e1e7c-Paper-Conference.pdf.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 21573–21612. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/4441469427094f8873d0fecb0c4e1cee-Paper-Datasets_and_Benchmarks.pdf.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Yangyang Yu, Zhiyuan Yao, Haohang Li, Zhiyang Deng, Yuechen Jiang, Yupeng Cao, Zhi Chen, Jordan W. Suchow, Zhenyu Cui, Rong Liu, Zhaozhuo Xu, Denghui Zhang, Koduvayur Subbalakshmi, Guojun Xiong, Yueru He, Jimin Huang, Dong Li, and Qianqian Xie. Fincon: A synthesized llm multi-agent system with conceptual verbal reinforcement for enhanced financial decision making. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 137010–137045. Curran Associates, Inc., 2024. doi: 10.52202/079017-4354. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/f7ae4fe91d96f50abc2211f09b6a7e49-Paper-Conference.pdf.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A Fault-Tolerant abstraction for In-Memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pp. 15–28, San Jose, CA, April 2012. USENIX Association. ISBN 978-931971-92-8. URL https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

Yusen Zhang, Ruoxi Sun, Yanfei Chen, Tomas Pfister, Rui Zhang, and Sercan Ö. Arı k. Chain of agents: Large language models collaborating on long-context tasks. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 132208–132237. Curran Associates, Inc., 2024. doi: 10.52202/079017-4202. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/ee71a4b14ec26710b39ee6be113d7750-Paper-Conference.pdf.

Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. GPT-4V(ision) is a generalist web agent, if grounded. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 61349–61385. PMLR, July 2024. URL https://proceedings.mlr.press/v235/zheng24e.html.

Zihan Zhou, Chong Li, Xinyi Chen, Shuo Wang, Yu Chao, Zhili Li, Haoyu Wang, Qi Shi, Zhixing Tan, Xu Han, Xiaodong Shi, Zhiyuan Liu, and Maosong Sun. LLM×MapReduce: Simplified long-sequence processing using large language models. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 27664–27678, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.1341. URL https://aclanthology.org/2025.acl-long.1341/.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language agents as optimizable graphs. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62743–62767. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/zhuge24a.html.

A Extended related work

Distributed execution architectures Many works (Dean & Ghemawat, 2004; Apache Software Foundation; Isard et al., 2007; Power & Li, 2010; Malewicz et al., 2010; Murray et al., 2011; Zaharia et al., 2012; Rocklin, 2015; Moritz et al., 2018) have studied ways to leverage distributed computing infrastructure for the efficient execution of data- and task-parallel workflows. Notably, MapReduce(Dean & Ghemawat, 2004) introduces a programming model and processing engine

capable of handling massive datasets in parallel across distributed, commodity hardware. The innovation of MapReduce spurred heightened research into distributed execution architectures that build on top of the idea and improve on it by using abstractions to model the workflows (Apache Software Foundation; Zaharia et al., 2012; Moritz et al., 2018).

LLM multi-agent systems LLM-based multi-agent systems are evolving along several threads. General purpose orchestration substrates expose conversational, role-based or graph-based abstractions to compose co-operating agents (Wu et al., 2024; Hong et al., 2024; Li et al., 2023; Zhuge et al., 2024), a line increasingly mirrored by deployed industry frameworks and systems (CrewAI, Inc.; OpenAI, 2026; Microsoft Corporation; Fourney et al., 2024). Building on these substrates, role-specialized teams emulate human workflows in domains such as SOP-driven software engineering (Qian et al., 2024; Dong et al., 2024). A separate thread improves reasoning through debate and consensus (Du et al., 2024; Liang et al., 2024), while others push towards scalability by dynamically generating agents without predefined SOPs (Wang et al., 2025b; Chen et al., 2024a; Hu et al., 2025).

LLM-powered data processing LOTUS (Patel et al., 2025) defines a notion of *semantic operators*, a declarative interface for semantic document manipulation similar to relational operators from database systems. DocETL (Shankar et al., 2025) uses LLM agents to rewrite and evaluate user-defined document processing pipelines. Unlike APWA, these systems require a human to author a data processing pipeline in a declarative language. In APWA, an agent determines the workflow based on the natural language task specification. Moreover, these works are restricted to document processing workflows in which the execution units consist only of LLM invocations. APWA in contrast supports arbitrary agentic subtasks as the execution unit.

Chain-of-Agents (CoA) (Zhang et al., 2024) considers long-context document processing for tasks like question answering, code completion, and summarization. CoA executes agents in sequence and therefore cannot benefit from speedup under parallelization. LLM×MapReduce (Zhou et al., 2025) performs long-context summarization with a single fixed parallel map pipeline over document chunks. It represents a fixed example of a workflow that APWA can discover autonomously.

Parallel tool calling LLMCompiler (Kim et al., 2024a) considers agents who emit tool call DAGs and implements a parallelized tool executor. Its parallelism is thus scoped to tool call execution. Large-scale parallelism cannot arise directly from this interface, as dispatching independent subtasks still requires emitting one tool call per subtask.

Serving and execution substrates Parrot (Lin et al., 2024) exposes dependencies between LLM requests through a *semantic variable* abstraction and optimizes over correlated requests. Autellix (Luo et al., 2025) schedules agentic programs as first-class units with program-level preemption and prioritization. SAGE (Liu et al., 2026) compiles LLM-augmented pipelines, including inference, vector retrieval, and agent state, into a jointly scheduled dataflow with declarative composition and fault recovery. Each of these systems targets optimizations sitting a layer below the agent, focusing on optimizing serving metrics such as time to first token. APWA operates above these systems and could feasibly use them to improve execution-time performance.

B APWA IMPLEMENTATION DETAILS

This appendix describes the implementation of each APWA component in detail. We cover the manager (Appendix B.1), workers (Appendix B.2), and executor (Appendix B.3); then planning and output contracts (Appendix B.4), data tables (Appendix B.6), and capability registry (Appendix B.7); and finally storage (Appendix B.8) and cluster management (Appendix B.9).

B.1 TASK MANAGER

The *Task Manager* (or just *Manager*) is the system component responsible for overseeing the life cycle of a single user task. It receives a task from the orchestrator enclosing the user query expressed in natural language and any pre-configured environment state in the form of references to external data located in the object store.

B.1.1 MANAGER AGENT

The core operative backbone of the Manager is its namesake *manager agent*. The logical role of the manager agent is conceptually similar to other hierarchically-organized multi-agent system architectures like Magentic-One (Fourney et al., 2024) and MegaAgent (Wang et al., 2025b), in that the manager agent is responsible for high-level planning, reasoning over intermediate global state, and coordinating subordinate subagents in order to make progress towards task completion. In addition to these roles, we also design the manager agent to be tightly integrated with our core system abstractions. These abstractions facilitate agent-based reasoning for parallel processing workflows over distributed computing resources.

Agent organization The per-round exploration phase, performed by the manager agent, involves three distinct steps. The first, most substantial step, is the iterative tool-calling loop which the agent uses to inspect and interact with the environment. During this step, the manager agent plans, reads execution state, creates agent presets, manipulates data tables, and stages subtasks to make progress towards task processing. Following this step, a non-tool-using *report generation* step is performed to extract a structured object describing any accomplishments, next steps, or critical blockers from the round. This intermediate step is primarily for observability and debugging purposes. Finally, a *delegation* step validates the coherence of the current agent trajectory against the task, plan, and output contract and determines if the proposed system transition (either subtask batch emission or task finalization) is appropriate. In the case that the transition is inappropriate, control is returned to the first step. This final step is motivated by the agent debate design pattern (Du et al., 2024; Liang et al., 2024), and is intended to help detect and resolve erroneous or premature state transitions.

LLM context formation To integrate our core abstractions with the manager agent, we must organize and format system state information into reasonably succinct textual representations to be fed into LLM-based components. Following existing agentic systems, we organize the system context as a chat-like transcript.

Starting with the manager agent's system prompt, we aim to provide a sufficiently coherent system model and catalog of domain concepts to the LLM. In particular, we describe the APWA system and the manager agent's operative role within it in the system prompt. Additionally, we describe all domain concepts, including the core APWA abstractions. Then, to better condition the model towards generating coherent parallelizable workflows, we provide several examples of generic parallel computation patterns and how to realize them.

B.1.2 MANAGER AGENT TOOLS

One of the crucial abstractions implemented in the task manager is the *comprehensive tool suite* designed to facilitate agent-based reasoning over distributed execution state. The tool suite conceptually consolidates all of our distributed agentic computation abstractions into a single unified programming interface connected to the Manager Agent. This tool suite extends the functionality of the base LLM agent to enable it to reason over parallelization patterns and inspect intermediate state in a way that carefully balances exploration with context efficiency. Table 5 lists all tools made available to the manager agent.

B.2 WORKER

B.2.1 LOGICAL EXECUTION MODES

APWA supports two execution modes, a *full agent* mode and *LLM-only* mode. When using the full agent model, a dedicated, isolated execution space is created for the purpose of processing that individual subtask. We implement these isolated execution spaces using a collection of Docker containers attached to a shared Docker network, but other implementations are possible. When developing APWA, we recognized that many subtasks can be realized in the form of a single LLM query with some data attached. For example, text summarization fits this paradigm. Therefore, to improve subtask throughput dynamically based on expected subtask complexity, we add an LLM-only execution path. This path does not create the full execution environment and implements subtask execution by making a single LLM query, formatting the agent request and all data resources inline with the LLM context. This allows us to process larger sub-batches within a single worker process,

Table 5: Tools used by the manager agent.

Tool	Description
Table inspection	
`list_tables`	List available tables.
`get_table_meta`	Return metadata for a table.
`preview_rows`	Show a paginated preview of rows from a table, optionally selecting columns and ordering results.
`get_row`	Retrieve one row from a table by row ID.
`filter_rows`	Return rows from a table that matching a predicate.
Table analytics	
`distinct_values`	List distinct values with counts for a field.
`value_counts`	Compute the top-k most frequent values for a field.
`summarize_numeric`	Compute summary statistics for one or more numeric fields.
`groupby_aggregate`	Perform grouped aggregation with count, count-distinct, sum, average, min, or max aggregations.
`sample_rows`	Draw a random sample of rows from a table.
Table Derivation	
`create_union_table`	Create a derived table by vertically concatenating multiple schema-compatible tables.
`create_filtered_table`	Create a derived table containing only rows from an input table that satisfy a predicate.
`create_projected_table`	Create a derived table containing only a specified subset of columns.
`create_joined_table`	Create a derived table by joining two input tables on specified key pairs.
`create_grouped_table`	Create a derived table by grouping rows and applying named first, count, or collect aggregations.
`rename_table`	Rename a table's display name without changing its data or lineage.
`archive_table`	Archive a table so it is hidden from ordinary table listings while preserving lineage.
`unarchive_table`	Restore an archived table to the active namespace, optionally assigning a display name.
`rename_columns`	Create a derived table with selected columns renamed.
`add_computed_columns`	Create a derived table with new computed columns.
`create_results_with_source`	Join a batch results table back to its discovered source table using lineage columns.
`drop_columns`	Create a derived table with specified columns removed.
Subtask staging tools	
`stage_single_subtask`	Add one literal subtask definition to the staging area.
`stage_dataset_subtask`	Stage a batch of subtasks by applying a template to every row of a dataset table.
`remove_staged_subtask`	Remove one staged subtask template.
`clear_staged_subtasks`	Clear all currently staged subtasks from the staging area.
`list_subtasks`	List executed subtasks with pagination and an optional status filter.
`get_subtask_result`	Fetch detailed run results, metrics, status, and artifacts for a specific subtask ID.
`get_artifact`	Retrieve artifact metadata and, optionally, truncated artifact content.
`list_artifacts`	List artifacts for the task, optionally filtered by subtask and optionally including previews.
Planning tools	
`write_plan`	Replace the manager plan with a new full plan payload.
`write_output_contract`	Replace the task output contract and reports whether the current dataset satisfies it.
Agent tools	
`create_agent_preset`	Create or updates an agent preset with a prompt and capability list.
`list_agent_presets`	List all currently available agent presets.
`delete_agent_preset`	Delete an agent preset by name.

which utilizes system resources more efficiently. The LLM-only execution path has the limitation that the worker agent cannot call tools or access a full computer environment, so it is only suitable for simple subtask types.

B.2.2 Worker execution environment

All full agent mode subtasks execute in a Docker-managed execution sandbox. The sandbox consists of several Docker containers connected over a shared Docker network. The subtask spec should contain all information (configurations, task details, setup metadata) required to specify which containers to start, how to configure them, and how to connect them.

The top-level component responsible for orchestrating the Docker environments is the "subtask manager." The subtask manager is responsible for reading the subtask spec, starting the necessary containers, and wiring up the network connections between them. The subtask manager also monitors the health of the containers and handles any necessary cleanup after the subtask execution is complete. The subtask manager executes directly on the host machine alongside other subtask managers. When using the ray orchestration layer, the subtask manager implementation serves as the entrypoint for a ray task.

The core container hosts a so-called "leader agent" which is responsible for overseeing the general lifecycle of the subtask. The leader agent directly interfaces with the subtask inputs and is equipped with tools that correspond with capabilities that originate from the capability registry and which are contained in the subtask spec. The final output of the subtask is written by the leader agent to

a pre-specified output path. This output will be schema-validated by the subtask manager before propagating the result to the originating task manager executor.

Within the context of a subtask execution fabric, a "helper agent" refers to a long-lived container (concurrent with the leader agent) that exposes a well-defined interface (details of which are included in the capability registry). These containers are created at subtask startup and persist for the entire duration of the subtask execution. These containers are therefore allowed to maintain state and execute long-running processes concurrently with the leader agent.

Helper agents are not permitted to communicate with each other, and only have a line of communication with the leader agent. Helper agents are also not permitted to spawn other helper agents or helper tools.

B.2.3 WORKER AGENT

To simplify connection with external data resources and improve resource efficiency, the worker environment does not directly open a connection with the object store backend, but rather connects to a node-level *object store proxy* (details in Appendix B.9).

B.3 EXECUTOR

B.3.1 AUTOMATIC RETRIES

A well-studied challenge faced by distributed computation frameworks is how to seamlessly address transient resource failures such as network partitions (Dean & Ghemawat, 2004; Zaharia et al., 2012; Moritz et al., 2018). A common solution is to simply retry subtasks that fail due to potentially transient errors. We adopt this approach at the subtask level: each subtask is attempted up to a maximum of 3 times. If any attempted execution succeeds, its output is persisted to stable storage and propagated through the batch result to the task manager. If all execution attempts fail, a logical subtask failure is forwarded instead to the task manager. This failure resolution is important for providing a clean logical layer to the manager agent.

B.4 PLANNING FOR PARALLELIZATION

A common design pattern in prior research (Fourney et al., 2024) investigates how to use dedicated planning phases to improve LLM agent performance on long-horizon tasks. To encourage parallel workflow reasoning, we adapt this design pattern and explicitly model the planning process as a first-class system abstraction. We realize two forms of planning abstractions, each realized as local manager state maintained by the manager agent. The first is a local planning structure that encodes high-level parallelization strategy in a flexible, generic format. The second encodes a formal task contract that self-imposes global state preconditions gating task termination.

Parallel planning structure We implement a simple planning abstraction to encourage parallelization-focused planning from the manager agent. Our parallel plan is designed to support APWA's goal of automated task decomposition into parallelizable workflows. Thus, we define the plan structure with explicit consideration of the manager's parallelization objectives. An APWA plan consists of two main fields: the *partition strategy* and a general-purpose *list of steps*. The partition strategy is intended to be populated with explicit data and task partition axes that eventually correspond to batches of subtasks that can be executed in parallel. The partition strategy is intended to represent a high-level parallelization strategy and be roughly fixed once established. To enable more granular planning controls, the planning structure also supports a general-purpose list of steps that identify the key procedural table manipulation and batch execution operations that will lead to task completion. These steps are annotated with agent-provided progress flags, allowing the manager agent to track progress towards individual subgoals.

To clarify the distinguishing conceptual properties of our plan structure, we contrast our planning abstraction with the planning abstraction from Magentic-One (Fourney et al., 2024). In addition to facilitating general-purpose planning, Magentic-One's two-tiered planning mechanism is structured primarily around the organization of *facts* anticipated and encountered during agent execution, determining the immediate next agent invocation, detecting unproductive agent loops, and deciding

appropriate termination conditions. This planning abstraction is well-suited for, e.g., knowledge gathering work, but does not facilitate dynamically-discovered parallelized workflows. In contrast, our planning abstraction explicitly models parallel task decomposition as a first-class consideration.

The manager agent can update the plan structure at any time during its exploration phase using a dedicated tool (see Appendix B.1.2).

Output contracts The manager's plan object primarily functions as a free-form scratch space whose contents are not formalized into any hard execution requirements. Its main purpose is to refine the manager agent's focus towards achieving the user task as well as to serve as a simple kind of agent memory. Depending on the task, it may also be possible to derive formal output specifications defining preconditions required for task completion. We implement this concept in the form of *output contracts*.

Our implementation of output contracts is grounded in APWA's data table abstraction. An output contract consists of a set of *table specifications*. A table specification defines a table that must exist before the task can be considered complete. Each table specification declares a name for the expected table, as well as (optionally) the column schema and number of rows the table is expected to have.

Agent-generated output contracts are used in a few different ways during system execution. Most immediately, the task contract (and intermediate progress towards its fulfillment) is formatted into the manager agent's context window. We hypothesize that conditioning LLM generation on this formatted intermediate progress focuses execution towards higher-utility agent trajectories. Second, when the manager agent signals task completion, we directly validate the manager table state against the contract in a rule-based manner. If the contract is unfulfilled, the task manager's control flow is returned to the manager agent to continue the exploration phase. Although simple, we find that simply encouraging the manager agent to formalize its task objectives as hard execution contracts can prevent premature task termination as well as improve output fidelity against user-specified formatting constraints.

Like the free-form parallelization plan, the manager agent can update the output contract at any point during its exploration phase using a dedicated tool (see Appendix B.1.2).

B.5 Subtask delegation

One of the novel challenges that arises from using LLM-based components to specify task decomposition into parallelizable workflows lies in how to simply *specify* that partitioning efficiently. Within standard (non-agentic) distributed execution engines, decomposition happens at a time scale typically orders of magnitude faster than the time taken for the subtasks to execute. In particular, it is relatively inexpensive for the programmer to specify thousands, or even millions of subtasks without noticeable overhead compared with the time taken to execute those subtasks. In contrast, with LLM-based delegation mechanisms, this dynamic is inverted. Defining a single subtask using a natural language specification can take on the order of seconds, introducing network and text generation overhead.

To resolve this issue, we introduce *subtask templates*. Subtask templates provide a way for LLM-based components to efficiently specify a large number of subtasks that can be executed in parallel.

B.6 Data tables

A core difficulty our architecture must grapple with is how to reason over large-scale data and distributed execution resources. For the types of tasks we consider, the data can be too large to fit on a single machine (making it impossible to store or examine all intermediate state on a single machine), and even the size of the metadata describing that data can quickly exhaust even frontier LLMs' context windows. The key logical abstraction we introduce to facilitate agent-based reasoning over large volumes of data is that of a *data table*.

A data table is a logical, read-only, finite sequence of records. A record is mapping from field names to typed values. All records in a dataset conform to a common *schema*, which fixes the set of typed fields. Types determine the admissible domain for field values.

Various operations can be used to produce new datasets from existing ones, forming a *table algebra*. These operations derive from the standard relational algebra operations like joins, intersections, unions, and filters.

Operators in this algebra are both *pure* (not modifying their inputs) and *deterministic*. Thus, sequences of nested operations form a declarative specification for how to construct a dataset from a collection of leaf datasets.

All tables in our representation are either *leaf tables* grounded in a metadata-laden stable storage, or are constructed by application of operators in the algebra to one or more dependencies. Thus, all non-leaf tables are fully specified by their *lineage*. A lineage (graph) is a directed acyclic graph whose nodes denote operation applications, whose leaves denote base datasets, and whose edges describe data dependencies. Knowledge of the lineage graph of a dataset is sufficient to reconstruct it from its root dependencies, provided the leaf datasets are available.

Our table abstraction has several key advantages. First, the table abstraction makes LLM-based reasoning about intermediate task states more tractable. Table metadata and summaries can be efficiently formatted to fit inside the context window of an LLM, while hiding high-volume data contents. An LLM-based component integrated with the table subsystem can efficiently specify highly data-parallel transformations, including local table manipulation as well

Second, immutability simplifies coordination with other distributed system components. In particular, consistency of replicated shared objects is achieved without the need for convoluted consensus protocols. This property also makes it easier to recompute missing objects using lineage metadata.

We note some explicit restrictions of our abstract data model. It does not allow for efficient fine-grained memory controls inexpressible with the supported algebraic manipulations.

Why not use an SQL interface? We introduce a hand-crafted agent-facing data table abstraction through which the agent interacts with data resources. Could we instead replace it with a SQL-like interface? We argue a naïve adoption of such an interface poses several limitations.

First, database query interfaces return *data* by default, whereas the premise of our design is that the LLM should reason over *metadata*. Second, SQL has no native counterpart to the table-to-template coupling, and would need to re-introduce such a mechanism. Third, general mutable state complicates distributed execution and makes data manipulation error-prone, whereas our tables are read-only with lineage tracking. This enables deterministic recomputation under worker retries and content-addressed caching, and structurally prevents workers from using the data plane as a coordination channel. Finally, free-form SQL is an open-ended generation space that is more difficult to verify and bound in general compared with our strongly-typed tool suite.

It may be feasible to modify an SQL-based interface to mitigate these issues, but we would argue that such a modification constitutes an original abstraction that is comparable to the one we already propose, re-expressed in a different notation.

B.7 Capability registry

The *Capability Registry* is a service exposing diverse software functionalities that can be used during subtask execution. These capabilities enable dynamic, composable functionalities for workers. We assume the capability registry is implemented as a horizontally-scalable, network-accessible, external service.

Capabilities are categorized based on their local execution model. *Service* capabilities inhabit long-running execution spaces that persist for the entire duration of its associated subtask's execution environment. Notably, service capabilities may perform work in parallel with the main subtask driver process. This property is useful for performing expensive, long-running computations concurrently with a subtask driver process or enabling state persistence. *Tool* capabilities are stateless interfaces whose execution duration is scoped only for a single invocation.

Entries in the Capability Registry consist of metadata (capability name, unique identifier, natural language description) and implementation location (e.g., a Docker image). In our application, capabilities are discovered by the task manager during the planning and analysis phases and used to perform specialized operations during subtask execution.

We distinguish our capability registry from existing tool discovery architectures to clarify its contributions. Our capability registry is substantially different from the Model Context Protocol (MCP) (Model Context Protocol a Series of LF Projects, LLC.), due to MCP's remote execution model. MCP exposes a function call interface through a standardized network API that facilitates tool discovery and tool invocation. This differs from our capability registry, which does not perform tool execution but instead points to an implementation of the tool software.

B.8 Data resources and state management

APWA assumes access to highly available and durable network-accessed storage resource. We leverage two storage interfaces: the *state store* and the *object store*.

State store The state store maintains all control state for the full set of tasks being processed by the APWA system. This includes task submissions, task manager state checkpoints, subtask definitions, subtask execution attempts, subtask outputs, and an append-only event log.

Object store The object store is specifically used for storage of large, unstructured data objects. This interface is particularly well-suited for passing data back and forth between subtask workers and the task manager and can handle text, binary, image, and other data formats.

Objects in the object store are *content-addressed* by a hash of their contents, meaning that URI contents are immutable. This greatly simplifies data coordination, as updates are only propagated by passing new *handles* to data. This property also implicitly restricts the ability of agents to use the data plane for communication, since they cannot send messages by modifying the contents pointed at by a given URI.

B.9 Cluster management

In this subsection, we discuss considerations for managing computational and data resources in the cluster computing environment.

B.9.1 Node resource management

Ray exposes logical resource controls that constrain the number of co-located subtasks executing on the same Ray node. These controls are flexible and allow constraints on resource use ranging from (virtual) hardware resources such as CPUs and GPUs to logical resources (for example, limiting port). We use this mechanism to control the maximum node-level subtask concurrency by configuring each subtask execution to consume 1 logical CPU resource. Each ray node connected to the cluster is registered with a number of (logical) cores, which is used to limit the maximum workload placed on that node.

B.9.2 Shared node services

Each node in APWA's computing cluster maintains a small set of necessary services to facilitate basic interactions with core APWA resources. One of these services is the *object store proxy*. The object store proxy sits interfaces between subtask execution environments and the true system-wide object store. Using a proxy service simplifies worker environment setup, as it allows workers to use a consistent, local network location for accessing data resources in the object store. Using a proxy service also allows caching of resources, which allows multiple co-located worker processes to access the same data resources at a lower amortized cost. Crucially, because objects are content-addressed, this cache does not introduce the usual cache staleness concerns.

C Extended experimental results

This appendix contains detailed descriptions, scoring rules, and experimental findings for the evaluations described in Section 4.

C.1 SUMMARYBENCH SPECIFICATION AND RESULTS

Task description Each task specifies an input table representing the items in a corpus of text. Each row in the input table contains keys describing the placement of each item within the hierarchical organization defining the corpus (act → scene → for *Romeo and Juliet*, part → act → scene for *Dynasts*, and volume → chapter → part → paragraph for *Roman*). The system must produce one summary table per level of the hierarchy, plus a final summary. Each summary table must contain keys indexing the corresponding level of the hierarchy and a key indicating the output (e.g., the `scene-summary` table contains keys {`act, scene, uri`}). Each successive level must be summarized from the summaries of the level immediately below it. The size of each corpus is given in Table 6.

Table 6: Description of datasets used for the hierarchical summarization task. Note some counts may appear nonstandard because we include front matter and other ancillary text.

Corpus	Level 1	Level 2	Level 3	Level 4	Size (kB)
Romeo and Juliet (R&J)	26	5	-	-	166
The Dynasts	133	21	5	-	882
Decline and Fall of the Roman Empire (Roman)	2588	296	71	6	10,203

Scoring The structural score is defined as the mean of six indicator variables per output table: the table exists, it has the correct number of rows, the table has the correct columns, each entry has a non-empty summary (URI resolution), and each summary is at most 250 words (word cap). We apply the same metrics for the baselines (which do not emit tables), suitably adapted for the different data representation. Here, "presence" of a table indicates the presence of the appropriate key in the generated JSON object emitted by the system. The semantic score defined as the ROUGE-1 F_1 between each produced summary and a reference summary, averaged per levels and then across levels. Missing summaries are scored as 0.

The reference summaries were generated using the hand-authored decomposition described below, using GPT-5.4 nano as the worker model.

Hand-authored decomposition The hand-authored decomposition processes the corpus hierarchy one layer at a time starting with the most granular layer. At each layer, it groups the items (either source documents, or outputs from the previous layer) by the next layer's key. The grouped items are processed in parallel with each subtask producing a summary of the documents passed as input. Any failed items are discarded at each layer (although this never occurs in our experiments). The process repeats until the corpus-level summary is produced.

Full results We provide detailed results for `SummaryBench` in Tables 7 to 9. The most failures occur with GPT-5.4 mini as the manager and GPT-5.4 nano as the worker.

C.2 PII-300K SPECIFICATION AND RESULTS

Task description We use the AI4Privacy PII-300k dataset (Ai4Privacy, 2024). Each task samples 64, 512, or 4,096 records at random from the dataset and formats them into data table The system must return a table with one sanitized text sample per record, in which every span of personal information is replaced by an appropriate placeholder. The placeholders belong to one of 27 categories defined by the dataset. The structural score is defined similarly to Appendix C.1, using table presence, output schema, and row count. The semantic score is defined as the macro-averaged F_1 of the detected spans over the 27 PII categories. We also measure the precision, recall, content similarity, and exact match rate against the reference text.

Hand-authored decomposition The hand-authored decomposition for `PII-300k` is a straightforward fan out followed by output extraction from each response.

Full results We report the full `PII-300k` results in Tables 10 to 12.

Table 7: Failure rates and wall-clock runtimes for `SummaryBench` across baselines and APWA variants. FR is computed over all selected trials. WC is computed over trials that produced output and reported as mean ± standard deviation.

	R&J		Dynasts		Roman	
Method	**FR**	**WC**	**FR**	**WC**	**FR**	**WC**
Direct	0%	19 ± 1	60%	76 ± 0	100%	$\perp$
Magentic-One	100%	$\perp$	100%	$\perp$	100%	$\perp$
MegaAgent	60%	472 ± 44	60%	248 ± 63	40%	579 ± 189
LangChain (flat)	0%	54 ± 16	20%	145 ± 20	20%	283 ± 307
LangChain (recursive)	0%	33 ± 6	40%	111 ± 7	40%	56 ± 43
Hand-written (sequential)	0%	89 ± 4	0%	497 ± 37	100%	$\perp$
Hand-written (asyncio)	0%	12 ± 1	0%	27 ± 7	0%	104 ± 20
Hand-written (Ray)	0%	12 ± 1	0%	32 ± 14	0%	85 ± 32
APWA (`mini×mini`)	0%	96 ± 19	0%	142 ± 20	0%	272 ± 53
APWA (`mini×nano`)	11%	85 ± 9	22%	142 ± 29	11%	281 ± 45
APWA (`5.4×mini`)	0%	168 ± 17	0%	235 ± 27	0%	372 ± 46
APWA (`5.4×nano`)	0%	174 ± 21	0%	234 ± 30	0%	390 ± 43

Table 8: Structural and semantic scores for `SummaryBench` across baselines and APWA variants. Scores are computed over trials that produced output and reported as mean ± standard deviation.

	R&J		Dynasts		Roman	
Method	**Str.**	**Sem.**	**Str.**	**Sem.**	**Str.**	**Sem.**
Direct	1.00 ± 0.00	0.43 ± 0.01	0.98 ± 0.02	0.21 ± 0.01	$\perp$	$\perp$
Magentic-One	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
MegaAgent	0.14 ± 0.14	0.04 ± 0.04	0.21 ± 0.12	0.02 ± 0.02	0.16 ± 0.23	0.02 ± 0.02
LangChain (flat)	1.00 ± 0.00	0.46 ± 0.04	0.90 ± 0.13	0.26 ± 0.02	0.12 ± 0.12	0.03 ± 0.03
LangChain (recursive)	0.99 ± 0.02	0.41 ± 0.06	0.91 ± 0.01	0.20 ± 0.03	0.22 ± 0.03	0.03 ± 0.02
Hand-written (sequential)	0.95 ± 0.02	0.49 ± 0.01	0.91 ± 0.02	0.50 ± 0.01	$\perp$	$\perp$
Hand-written (asyncio)	0.97 ± 0.01	0.50 ± 0.01	0.90 ± 0.01	0.49 ± 0.01	0.88 ± 0.00	0.40 ± 0.00
Hand-written (Ray)	0.92 ± 0.00	0.47 ± 0.01	0.90 ± 0.01	0.50 ± 0.01	0.88 ± 0.00	0.39 ± 0.00
APWA (`mini×mini`)	0.95 ± 0.08	0.42 ± 0.14	0.97 ± 0.03	0.45 ± 0.06	0.91 ± 0.13	0.23 ± 0.16
APWA (`mini×nano`)	0.95 ± 0.04	0.43 ± 0.04	0.87 ± 0.13	0.32 ± 0.17	0.91 ± 0.14	0.30 ± 0.13
APWA (`5.4×mini`)	1.00 ± 0.00	0.53 ± 0.01	1.00 ± 0.00	0.45 ± 0.03	0.98 ± 0.03	0.37 ± 0.02
APWA (`5.4×nano`)	0.94 ± 0.03	0.44 ± 0.01	0.92 ± 0.05	0.42 ± 0.02	0.87 ± 0.01	0.40 ± 0.01

Table 9: `SummaryBench` detailed metrics by APWA configuration, corpus, and hierarchy layer. Values are averaged over APWA trials that produced the corresponding layer output.

			ROUGE			Structural		
Config	**Corpus**	**Layer**	**R-1**	**R-2**	**R-L**	**Present**	**URI res.**	**Word cap**
APWA (`mini×mini`)	R&J	Scene	0.35	0.12	0.23	1.00	0.86	0.60
		Act	0.41	0.13	0.24	1.00	0.86	0.86
		Full	0.51	0.14	0.32	1.00	1.00	1.00
	Dyn.	Scene	0.41	0.14	0.25	1.00	0.86	0.71
		Act	0.47	0.12	0.24	1.00	1.00	0.94
		Part	0.43	0.11	0.23	1.00	1.00	0.94
		Full	0.48	0.12	0.22	1.00	1.00	0.86
	Roman	Paragraph	0.36	0.11	0.22	1.00	0.86	0.86
		Part	0.22	0.03	0.10	1.00	0.71	0.71
		Chapter	0.21	0.03	0.09	1.00	0.71	0.69
		Volume	0.20	0.02	0.08	1.00	0.71	0.69
		Full	0.17	0.02	0.08	1.00	0.71	0.57
APWA (`mini×nano`)	R&J	Scene	0.40	0.11	0.23	1.00	1.00	0.82
		Act	0.39	0.10	0.21	1.00	1.00	0.56
		Full	0.43	0.10	0.23	0.89	0.89	0.67
	Dyn.	Scene	0.44	0.14	0.26	1.00	1.00	0.62
		Act	0.34	0.07	0.16	1.00	0.89	0.29
		Part	0.25	0.05	0.12	0.89	0.78	0.58
		Full	0.22	0.04	0.10	0.67	0.67	0.22
	Roman	Paragraph	0.47	0.16	0.31	1.00	1.00	0.88
		Part	0.31	0.04	0.14	1.00	1.00	0.63
		Chapter	0.23	0.02	0.10	0.89	0.89	0.64
		Volume	0.20	0.02	0.08	0.78	0.78	0.65
		Full	0.20	0.01	0.08	0.78	0.78	0.67
APWA (`5.4×mini`)	R&J	Scene	0.53	0.18	0.34	1.00	1.00	1.00
		Act	0.48	0.15	0.26	1.00	1.00	1.00
		Full	0.58	0.18	0.36	1.00	1.00	1.00
	Dyn.	Scene	0.52	0.17	0.31	1.00	1.00	1.00
		Act	0.46	0.12	0.24	1.00	1.00	0.96
		Part	0.42	0.10	0.22	1.00	1.00	0.98
		Full	0.41	0.09	0.20	1.00	1.00	1.00
	Roman	Paragraph	0.47	0.14	0.29	1.00	1.00	1.00
		Part	0.39	0.06	0.17	1.00	1.00	0.96
		Chapter	0.36	0.04	0.16	1.00	1.00	0.93
		Volume	0.31	0.03	0.13	1.00	1.00	0.93
		Full	0.32	0.02	0.14	1.00	1.00	0.67
APWA (`5.4×nano`)	R&J	Scene	0.44	0.12	0.25	1.00	1.00	0.95
		Act	0.39	0.10	0.21	1.00	1.00	0.59
		Full	0.49	0.11	0.25	1.00	1.00	0.44
	Dyn.	Scene	0.50	0.16	0.29	1.00	1.00	0.68
		Act	0.43	0.09	0.20	1.00	1.00	0.32
		Part	0.38	0.08	0.18	1.00	1.00	0.60
		Full	0.37	0.07	0.16	1.00	1.00	0.56
	Roman	Paragraph	0.51	0.16	0.32	1.00	1.00	0.75
		Part	0.40	0.06	0.17	1.00	1.00	0.12
		Chapter	0.37	0.04	0.15	1.00	1.00	0.10
		Volume	0.35	0.03	0.14	1.00	1.00	0.20
		Full	0.35	0.03	0.14	1.00	1.00	0.00

Table 10: Failure rates and wall-clock runtimes for `PII-300k` across baselines and APWA variants. FR is computed over all selected trials. WC is computed over trials that produced output and reported as mean $\pm$ standard deviation.

	PII-64		PII-512		PII-4096	
Method	**FR**	**WC**	**FR**	**WC**	**FR**	**WC**
Direct	0%	37 ± 4	0%	41 ± 29	100%	$\perp$
Magentic-One	100%	$\perp$	100%	$\perp$	80%	91 ± 0
MegaAgent	60%	390 ± 266	80%	25	70%	372 ± 488
Hand-written (asyncio)	0%	2	0%	8	0%	61
Hand-written (Ray)	0%	4	0%	6	100%	$\perp$
APWA (`mini×mini`)	0%	35 ± 4	0%	67 ± 52	0%	221 ± 119

Table 11: Structural and semantic scores for `PII-300k` across baselines and APWA variants. Scores are computed over trials that produced output and reported as mean ± standard deviation.

	PII-64		PII-512		PII-4096	
Method	**Str.**	**Sem.**	**Str.**	**Sem.**	**Str.**	**Sem.**
Direct	1.00 ± 0.00	0.78 ± 0.05	0.75 ± 0.00	0.16 ± 0.10	$\perp$	$\perp$
Magentic-One	$\perp$	$\perp$	$\perp$	$\perp$	1.00 ± 0.00	0.18 ± 0.08
MegaAgent	0.38 ± 0.38	0.00 ± 0.00	0.00	0.00	0.25 ± 0.35	0.00 ± 0.00
Hand-written (asyncio)	1.00	0.84	1.00	0.84	1.00	0.83
Hand-written (Ray)	1.00	0.85	1.00	0.84	$\perp$	$\perp$
APWA (`mini×mini`)	1.00 ± 0.00	0.76 ± 0.06	1.00 ± 0.00	0.77 ± 0.04	0.90 ± 0.30	0.54 ± 0.36

Table 12: `PII-300k` detailed structural and semantic metrics by APWA configuration and document count. Values are averaged over APWA trials that produced output.

Config	**Size**	**Struct.**	**PII Prec.**	**PII Recall**	**PII F1**	**Content Sim.**	**Exact Match**
APWA (`mini×mini`)	PII-64	1.00	0.83	0.76	0.79	0.89	0.35
	PII-512	1.00	0.84	0.76	0.80	0.89	0.35
	PII-4096	0.90	0.59	0.53	0.56	0.69	0.24

C.3 SCHEMABENCH SPECIFICATION AND RESULTS

Task description `SchemaBench` (Bai et al., 2024) provides a dataset of documents formatted as LaTeX, XML, CSV, or HTML. The task is to process each record in a given dataset to produce a structured JSON record containing the same information. It consists of four suites, each of which we treat as a separate APWA task: CHEMISTRY (chemistry tables), MLTABLES (machine learning result tables), DISCOMAT (materials science tables), and SWDE (web pages).

Scoring The structural score is defined as the mean of three boolean indicators: the output is present, the number of output entries matches the expected number, and every record has the prescribed output schema. The semantic score follows the metric used by the original work, described here. Each suite computes the word-level F_1 score between the predicted and reference values after normalizing outputs (removing LaTeX, unit symbols, and punctuation, lowercase, and whitespace). CHEMISTRY and MLTABLES pair The semantic score is then the macro F_1 score for CHEMISTRY, MLTABLES, and SWDE, and the micro-F_1 score for DISCOMAT.

Full results We report full results for `SchemaBench` experiments in Tables 13 to 15.

Table 13: Failure rates and wall-clock runtimes for `SchemaBench` across baselines and APWA variants. FR is computed over all selected trials. WC is computed over trials that produced output and reported as mean ± standard deviation.

	CHEMISTRY		MLTABLES		DISCOMAT		SWDE	
Method	**FR**	**WC**	**FR**	**WC**	**FR**	**WC**	**FR**	**WC**
Direct	0%	350 ± 44	100%	$\perp$	100%	$\perp$	100%	$\perp$
Magentic-One	80%	95	60%	79 ± 52	100%	$\perp$	60%	88 ± 47
MegaAgent	60%	141 ± 53	100%	$\perp$	40%	278 ± 202	80%	1222
APWA (`mini×mini`)	10%	62 ± 17	10%	137 ± 25	10%	103 ± 22	10%	104 ± 39
APWA (`mini×nano`)	10%	75 ± 14	10%	191 ± 57	0%	101 ± 49	0%	95 ± 20
APWA (`5.4×mini`)	0%	95 ± 10	0%	183 ± 22	0%	147 ± 13	10%	125 ± 11
APWA (`5.4×nano`)	10%	115 ± 8	10%	231 ± 37	10%	187 ± 27	10%	128 ± 17

C.4 REPORTBENCH SPECIFICATION AND RESULTS

Task description The goal of the benchmark is to compose a report (100-200 words) on a manually curated set of topics drawn from diverse domains. It is designed to show that APWA is not only capable of dynamically identifying the capability presets required for a task but is also able to utilize the corresponding agent from the registry at scale. Topics are divided into small (10), moderate

Table 14: Structural and semantic scores for `SchemaBench` across baselines and APWA variants. Scores are computed over trials that produced output and reported as mean ± standard deviation.

	CHEMISTRY		MLTABLES		DISCOMAT		SWDE	
Method	**Str.**	**Sem.**	**Str.**	**Sem.**	**Str.**	**Sem.**	**Str.**	**Sem.**
Direct	0.67 ± 0.00	0.53 ± 0.09	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
Magentic-One	1.00	0.46	1.00 ± 0.00	0.05 ± 0.00	$\perp$	$\perp$	1.00 ± 0.00	0.26 ± 0.08
MegaAgent	0.33 ± 0.33	0.00 ± 0.00	$\perp$	$\perp$	0.22 ± 0.31	0.00 ± 0.00	0.67	0.00
APWA (`mini×mini`)	1.00 ± 0.00	0.71 ± 0.25	1.00 ± 0.00	0.53 ± 0.12	0.96 ± 0.11	0.66 ± 0.24	1.00 ± 0.00	0.86 ± 0.22
APWA (`mini×nano`)	1.00 ± 0.00	0.58 ± 0.18	1.00 ± 0.00	0.47 ± 0.13	0.72 ± 0.12	0.65 ± 0.04	1.00 ± 0.00	0.81 ± 0.25
APWA (`5.4×mini`)	1.00 ± 0.00	0.79 ± 0.03	1.00 ± 0.00	0.57 ± 0.03	1.00 ± 0.00	0.80 ± 0.02	1.00 ± 0.00	0.94 ± 0.00
APWA (`5.4×nano`)	1.00 ± 0.00	0.76 ± 0.03	1.00 ± 0.00	0.55 ± 0.03	0.74 ± 0.14	0.70 ± 0.05	1.00 ± 0.00	0.92 ± 0.01

Table 15: `SchemaBench` detailed structural and classification metrics by APWA configuration and schema extraction suite. Values are averaged over APWA trials that produced output.

			Micro			Macro		
Config	**Suite**	**Struct.**	**P**	**R**	**F1**	**P**	**R**	**F1**
APWA (`mini×mini`)	CHEMISTRY	1.00	0.70	0.79	0.74	0.69	0.76	0.71
	MLTABLES	1.00	0.67	0.55	0.58	0.59	0.52	0.53
	DISCOMAT	0.96	0.66	0.66	0.66	N/A	N/A	N/A
	SWDE	1.00	N/A	N/A	N/A	N/A	N/A	0.86
APWA (`mini×nano`)	CHEMISTRY	1.00	0.69	0.64	0.66	0.59	0.62	0.58
	MLTABLES	1.00	0.59	0.45	0.49	0.57	0.44	0.47
	DISCOMAT	0.72	0.67	0.63	0.65	N/A	N/A	N/A
	SWDE	1.00	N/A	N/A	N/A	N/A	N/A	0.81
APWA (`5.4×mini`)	CHEMISTRY	1.00	0.77	0.87	0.82	0.79	0.84	0.79
	MLTABLES	1.00	0.67	0.63	0.65	0.58	0.57	0.57
	DISCOMAT	1.00	0.79	0.81	0.80	N/A	N/A	N/A
	SWDE	1.00	N/A	N/A	N/A	N/A	N/A	0.85
APWA (`5.4×nano`)	CHEMISTRY	1.00	0.77	0.81	0.79	0.76	0.82	0.76
	MLTABLES	1.00	0.65	0.56	0.60	0.61	0.53	0.55
	DISCOMAT	0.74	0.69	0.72	0.70	N/A	N/A	N/A
	SWDE	1.00	N/A	N/A	N/A	N/A	N/A	0.92

Table 16: Failure rates and wall-clock runtimes for `TranslationBench`. FR = failure rate, WC = wall-clock time (seconds).

	1 doc		16 docs		32 docs		64 docs	
Method	**FR (↓)**	**WC (s) (↓)**	**FR (↓)**	**WC (s) (↓)**	**FR (↓)**	**WC (s) (↓)**	**FR (↓)**	**WC (s) (↓)**
APWA (`mini×nano`)	0%	56 ± 7	12%	140 ± 25	0%	310 ± 136	0%	383 ± 94
APWA (`5.4×nano`)	0%	85 ± 33	0%	296 ± 155	0%	342 ± 150	0%	437 ± 138

(20) and large (100) splits. The benchmark will require the need of an agent with web browsing capabilities for which we use the Websurfer agent from Magentic-One.

We score APWA along two dimensions: utility (whether the system produces a correct output) and cost (wall-clock runtime). We decompose utility into two submetrics, structural (whether the outputs are correctly formatted) and semantic (whether the outputs are correct). The structural score measures whether APWA returns the expected number of (topic, report) pairs. The semantic score evaluates each pair on 2 attributes - whether the report addresses the requested topic and whether the content is substantive, factual information rather than filler or off-topic. We use LLM-as-a-judge to assess these attributes, a pair receives a score of 1.0 if both attributes hold, 0.5 if only one holds and 0.0 otherwise. The final semantic utility is a mean over all pairs.

Full results On the small split (10 topics), APWA completes the task in 2 min 23 s while maintaining a perfect score on both structural and semantic utility, successfully producing 10 relevant and factual reports. On the moderate split (20 topics), runtime increases to 2 min 37 s but APWA achieves a perfect structural score while yielding a semantic score of 0.975, with 19 out of 20 topics having a relevant and factual report and 1 deemed as relevant but not substantive. On the large split (100 topics), runtime grows to 10 min; APWA again achieves a perfect structural score and a semantic utility of 0.995, with 99 of 100 topics yielding a relevant and factual report and 1 deemed relevant but lacking substantive content. Notably, the runtime scaling is sublinear, increasing the workload from 10 to 100 topics (a $10\times$ increase) results in only a $4.2\times$ increase in the runtime. This suggests that the execution overhead is progressively amortized across larger batches, indicating favorable scalability for the task.

C.5 TranslationBench specification and results

Task description We use the FLORES+ dataset Costa-jussà et al. (2024), created for evaluating multilingual machine translation. The task is to translate each item in a table of *source sentences*, written in English (`eng_Latn`), into each of a collection of different *language varieties* supplied in a second input table. Target language varieties (consisting of language name, ISO 639-3 code and ISO 15924 script) are drawn from all available FLORES+ configurations. We vary the number of source sentences over 1, 16, 32, 64 and in every case use the full set of 213 target language varieties, producing output grids of size up to $64 \times 213 = 13,632$ items.

Scoring The structural score is defined as the mean of four boolean indicators: the output table is present, it has the expected number of rows, it has the required schema, and each empty contains a non-empty translation. The semantic score is chrF++ (Popović, 2017), following the benchmark, computing using the sacreBLEU package (Post, 2018). We also report spBLEU following the original paper (Costa-jussà et al., 2024).

Full results Full results are reported in Tables 16 to 18.

C.6 Scalability experiment

Setup We measure how the executor scales with cluster size. We provision Ray clusters of 1, 2, 4, 8, 16, and 30 AWS EC2 `m6i.4xlarge` instances (16 vCPUs, 64 GB memory each) and run the full APWA service stack (state store, object store, and capability registry) in each configuration. Each node sets a maximum Ray task concurrency limit of 16.

Table 17: Structural and semantic scores for `TranslationBench`. The semantic score is the mean chrF++ normalized to $[0, 1]$. Scores are computed over trials that produced output and reported as mean $\pm$ standard deviation. Str. = structural score, Sem. = semantic score.

	1 doc		16 docs		32 docs		64 docs	
Method	**Str. (↑)**	**Sem. (↑)**	**Str. (↑)**	**Sem. (↑)**	**Str. (↑)**	**Sem. (↑)**	**Str. (↑)**	**Sem. (↑)**
APWA (`mini×nano`)	1.00 ± 0.00	0.37 ± 0.05	0.96 ± 0.09	0.32 ± 0.12	1.00 ± 0.00	0.37 ± 0.01	0.81 ± 0.33	0.33 ± 0.12
APWA (`5.4×nano`)	1.00 ± 0.00	0.38 ± 0.05	0.97 ± 0.08	0.33 ± 0.12	0.94 ± 0.11	0.33 ± 0.08	0.97 ± 0.08	0.33 ± 0.12

Table 18: `TranslationBench` detailed metrics by APWA configuration and input size. spBLEU and chrF++ are the metrics used by the original NLLB translation work. Languages is the number of target language varieties covered. Pairs produced and scored count (sentence, language) outputs. Values are averaged over trials that produced output.

Config	**Size**	**Struct.**	**spBLEU**	**chrF++**
APWA (`mini×nano`)	1 doc	1.00	12.67	37.33
	16 docs	0.96	10.23	31.74
	32 docs	1.00	12.82	37.20
	64 docs	0.81	11.51	32.76
APWA (`5.4×nano`)	1 doc	1.00	12.72	37.63
	16 docs	0.97	10.56	32.47
	32 docs	0.94	11.00	32.66
	64 docs	0.97	11.58	33.13

We measure execution throughput for both LLM-direct and full execution modes by performing the entire subtask batch execution phase, including dispatching the batch to APWA's executor, submitting the individual subtasks as Ray tasks, collecting the results, and writing records to APWA's state and object stores.

Subtasks for the LLM-direct path generate a response to a single query. Subtasks for the full path create the full virtualized environment with a single agent capability and performs a single scripted capability invocation.

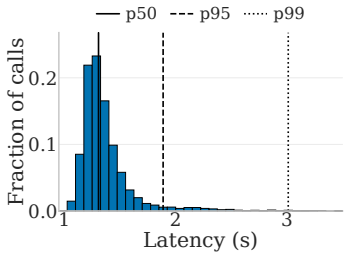


Figure 4: Sampled GPT-5.4 nano latency distribution.

To reduce evaluation costs, we mock LLM calls with a node-level service. We simulate the LLM call latency distribution by bootstrap sampling latencies collected from 8,192 independent GPT-5.4 nano LLM calls (Figure 4).

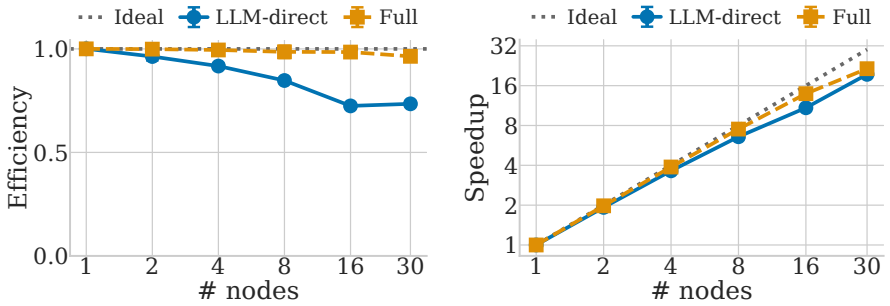
Scoring We measure *speedup* and *efficiency* under both *strong* and *weak* scaling. Strong scaling fixes a total workload size (389,120 LLM-only subtasks and 2,048 full subtasks). Weak scaling varies the workload size with cluster size (48,640 LLM-only and 256 full subtasks per node). Throughput for each trial is defined as $\mathrm{thr}_n = S_n/T_n$, where S_n is the number of processed subtasks and T_n is the end-to-end wall clock runtime for the n-node cluster. Speedup is defined as the ratio $\mathrm{spd}(n) = \mathrm{thr}(n)/\mathrm{thr}(1)$. Efficiency is defined as $\mathrm{eff}(n) = \mathrm{spd}(n)/n$.

Results Metrics are reported in Table 19 and Figure 5. We run 7 trials for each configuration. Throughput increases consistently up to 30 nodes in both settings. At 30 nodes, the LLM-direct mode reaches over 4,100 subtasks per second, while the full execution mode sits slightly below 30 subtasks per second. The LLM-direct mode is able to accommodate much higher throughput due to the smaller subtask complexity (a single LLM invocation) and sub-batching in the Ray task (each Ray task performs 64 APWA subtasks).

The efficiency for the LLM-direct mode is lower than that for the full execution mode. This is due to larger I/O overhead, as individual records for each subtask must be written to the state store and object stores. Scaling in this case could be improved further by horizontally scaling the state store (we use a single-node deployment with no sharding) and by modifying the persisted output representation (batching writes). The low throughput for the full execution mode does not stress this bottleneck. The higher full subtask latency is due to higher overhead from creating the sandboxed environments, which we expect would become marginal for larger subtask workloads.

Table 19: Scaling of APWA executor on AWS EC2, using 1 to 30 `m6i.4xlarge` nodes (16 vCPU each). Weak scaling fixes per-node workload. Strong scaling fixes the total workload. Throughput is measured as subtasks per second, reported as mean ± standard deviation over 7 trials. Speedup and efficiency are relative to the same mode's single-node throughput. Thr. = throughput, Eff. = efficiency.

		Weak scaling			Strong scaling		
Mode	**Nodes**	**Thr. (/s)**	**Speedup**	**Eff.**	**Thr. (/s)**	**Speedup**	**Eff.**
LLM-DIRECT	1	188 ± 5	1.00	1.00	194 ± 3	1.00	1.00
	2	361 ± 7	1.93	0.96	372 ± 4	1.92	0.96
	4	687 ± 8	3.67	0.92	704 ± 8	3.63	0.91
	8	1270 ± 9	6.77	0.85	1273 ± 12	6.58	0.82
	16	2173 ± 9	11.59	0.72	2108 ± 21	10.88	0.68
	30	4130 ± 22	22.03	0.73	3771 ± 26	19.47	0.65
FULL	1	1.03 ± 0.02	1.00	1.00	1.09 ± 0.00	1.00	1.00
	2	2.06 ± 0.04	2.00	1.00	2.16 ± 0.01	1.98	0.99
	4	4.10 ± 0.09	3.98	0.99	4.24 ± 0.05	3.88	0.97
	8	8.13 ± 0.23	7.88	0.99	8.19 ± 0.13	7.51	0.94
	16	16.24 ± 0.25	15.75	0.98	15.18 ± 0.46	13.92	0.87
	30	29.81 ± 0.72	28.92	0.96	23.44 ± 1.35	21.49	0.72



(a) Weak-scaling efficiency.

(b) Strong-scaling speedup.

Figure 5: APWA executor scalability. Figure 5a shows the weak scaling efficiency curve, which indicates APWA throughput at saturation. Figure 5b shows the strong scaling speedup curve, which shows speedup on a fixed task size.