
A CPU-CENTRIC PERSPECTIVE ON AGENTIC AI

Ritik Raj^{† 1} Hong Wang² Tushar Krishna¹

ABSTRACT

Agentic AI frameworks add a decision-making orchestrator embedded with external tools, including web search, Python interpreter, contextual database, and others, on top of monolithic LLMs, turning them from passive text oracles into autonomous problem-solvers that can plan, call tools, remember past steps, and adapt on the fly.

This paper aims to characterize and understand the system bottlenecks introduced by agentic AI workloads from a largely overlooked CPU-centric perspective. We first systematically characterize Agentic AI on the basis of orchestrator/decision making component, inference path dynamics and repetitiveness of the agentic flow which directly influences the system-level performance. Thereafter, based on the characterization, we choose five representative agentic AI workloads- Haystack RAG, Toolformer, ChemCrow, Langchain and SWE-Agent to profile latency, throughput and energy metrics and demystify the significant impact of CPUs on these metrics relative to GPUs. We observe that - ❶ Tool processing on CPUs can take up to 90.6% of the total latency; ❷ Agentic throughput gets bottlenecked either by CPU factors - coherence, synchronization and over-subscription of cores or GPU factors - main memory capacity and bandwidth; ❸ CPU dynamic energy consumes up to 44% of the total dynamic energy at large batch sizes. Based on the profiling insights, we present two key optimizations- ❶ CPU and GPU-Aware Micro-batching (**CGAM**) and ❷ Mixed Agentic Workload Scheduling (**MAWS**) for homogeneous and heterogeneous agentic workloads respectively to demonstrate the potential to improve the performance, efficiency, and scalability of agentic AI. We achieve up to $2.1\times$ and $1.41\times$ P50 latency speedup compared to the multi-processing benchmark for homogeneous and heterogeneous agentic workloads respectively.

1 INTRODUCTION

Large Language Models (LLMs) (Zhao et al., 2023) have leapfrogged the advancements in Artificial Intelligence (AI) for a plethora of applications, including vision (Wang et al., 2023; Zhou et al., 2024), healthcare (Thirunavukarasu et al., 2023; Bedi et al., 2025), science (Telenti et al., 2024; Jablonka et al., 2024), education (Gan et al., 2023; Wang et al., 2024), autonomous driving (Fu et al., 2024; Cui et al., 2023), and so on. However, they face challenges including context-agnosticism (Berglund et al., 2023), hallucinations (Maynez et al., 2020) and the lack of real-time information (Komeili et al., 2021; Ouyang et al., 2023).

Unlike monolithic LLMs that process tasks through single-pass inference, agentic AI (Shavit et al., 2023) frameworks (Schick et al., 2023; Singh et al., 2025) orchestrate multiple components including tool use, memory modules, and iterative reasoning loops to achieve superior performance. Recent benchmarks reveal that agentic frameworks such as ReAct (Yao et al., 2023) achieve 27% higher success rates on ALFWorld (Shridhar et al., 2020) tasks and 34% improvement on WebShop (Yao et al., 2022) compared to

equivalent-sized monolithic models, while AutoGPT (Yang et al., 2023) and BabyAGI (Nakajima) demonstrate $2\text{-}3\times$ better performance on long-horizon planning tasks despite using smaller base models. The performance advantages are particularly pronounced for domains requiring external knowledge integration and iterative refinement. For example, WebGPT (Nakano et al., 2021) shows that 7B parameter models can match or exceed the performance of 70B monolithic models on knowledge-intensive tasks, with achieving 64.1% accuracy on TruthfulQA (Lin et al., 2021) compared to 59.3% for GPT-3 (Brown et al., 2020) despite being $25\times$ smaller.

The choice of AI models for agentic AI workloads is an active area of research. Small Language Models (SLMs) are a good fit for agentic AI (Belcak et al., 2025) because agents thrive on fast, iterative perceive-plan-act loops, and privacy-preserving local execution. Many agent competencies are externalized: tool use and retrieval can offload computation and factual recall, reducing reliance on parametric capacity while preserving task performance, a setting in which SLMs including GPT-J 6B (Wang & Komatsuzaki, 2021) can be effective as shown in (Schick et al., 2023) and even beat much larger monolithic LLMs including OPT 66B (Zhang et al., 2022) and GPT-3 175B (Brown et al., 2020). Furthermore, recent studies (Gunasekar et al., 2023; Abdin et al., 2024)

[†]Work done during an internship at Intel. ¹Georgia Institute of Technology, Atlanta, GA, USA ²Intel, Santa Clara, CA, USA. Correspondence to: Ritik Raj <ritik.raj@gatech.edu>, Hong Wang <hong.wang@intel.com>.

shows sub-10B models achieving competitive capability on MMLU (Hendrycks et al., 2020) and MT-bench (Zheng et al., 2023) benchmarks as compared to LLMs including GPT-3.5 when trained with high-quality data and efficient architectures. However, SLMs often falter on long-horizon planning, scientific tasks and multi-tool orchestration. For example, GPT-4.1 scored 54.6% on SWE-bench (Jimenez et al., 2023), while fine-tuned open SLM SWE-Llama-7B (Princeton NLP Group, 2023) scored 3.0% and the 13B variant scored 4.0% (even with oracle retrieval). Therefore, in this work, we choose LLMs for scientific and coding tasks while SLMs for relatively simpler tasks.

Although AI models run mostly on GPUs or accelerators, CPUs are used in tool processing including Python or Bash execution, web search, URL fetching, lexical summarization (Erkan & Radev, 2004) and Exact Nearest Neighbor Search (ENNS) on large databases to name a few. While prior approaches on AI efficiency aggressively focused on GPU kernels and KV-cache scheduling, agentic AI brings a plethora of CPU-centric tools in the execution pipeline. A recent research (Quinn et al., 2025) shows ENNS account for more than 75% of the end-to-end latency on a 200 GB document corpus for a RAG (Retrieval Augmented Generation) workload consisting of Llama-3-70B (Dubey et al., 2024) model for generation. Another research (Jiang et al., 2025) shows that hyper-scale retrieval can pose a significant bottleneck in RAG pipelines for SLMs and higher retrieval quality. Furthermore, (Patel et al., 2024) argued that web agent benchmarks like WebArena (Zhou et al., 2023b) are computationally intensive due to latency from real-time web interactions, where LLM actions can’t be batched. (Xu et al., 2024) shows that tool partial execution can reduce request completion latency by up to 38.8% revealing that tool execution is a significant part of the end-to-end latency. To address the new CPU-centric perspective and optimize for agentic AI, this paper makes three major contributions:

System level Characterization: We introduce three fundamental and orthogonal categorization bases (Section 3) - orchestrator-based (LLM, host), agentic flow/repetitiveness (single-step, few-step, multi-step) and agentic path (static, dynamic) that comprehensively capture the computational and architectural diversity of agentic AI systems. These bases directly influence the system-level metrics of both CPUs and GPUs including performance and power.

Demystify CPU bottlenecks: To optimize for agentic AI, we profile full system including CPU and GPU for latency timeline (Section 4.2), batch throughput (Section 4.3) and energy (Section 4.4). We observe- ❶ Tool processing on CPUs can take up to 90.6% of the total latency. ❷ Agentic throughput is bottlenecked either by CPU factors - number of cores, coherence and synchronization or GPU factors - main memory capacity and bandwidth. ❸ CPU dynamic

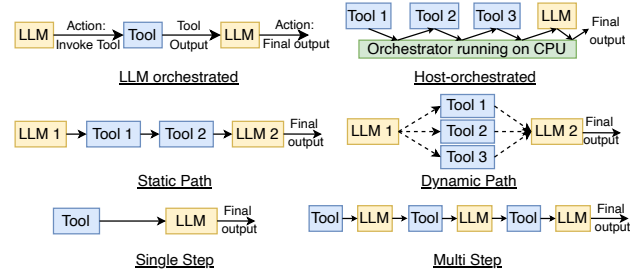


Figure 1. Characterization of agentic AI workloads on the basis of (a) Orchestrator (LLM and Host) (b) Agentic Path (Static and Dynamic) and (c) Repetitiveness (Single-step and Multi-step)

energy consumes up to 44% of the total dynamic energy at large batch sizes. To the best of our knowledge, this is the first work to demystify agentic AI CPU bottlenecks comprehensively across the three evaluation metrics.

Scheduling optimizations: Based on the profiling insights, we present two key scheduling optimizations- ❶ CPU and GPU-Aware Micro-batching (CGAM - Section 5.1) and ❷ Mixed Agentic Workload Scheduling (MAWS - Section 5.2) for homogeneous and heterogeneous agentic workloads respectively. We achieve up to $2.1\times$ and $1.41\times$ P50 latency speedup compared to multi-processing benchmark for homogeneous and heterogeneous agentic workloads respectively.

2 BACKGROUND

2.1 Tool-augmented Agentic AI

Tool-based agents couple language models with external tools and APIs to plan, invoke actions, and incorporate results in a closed loop, enabling goal-directed behavior beyond single-shot text generation. Foundational approaches such as ReAct (Yao et al., 2023) interleave reasoning traces with tool calls, while Toolformer (Schick et al., 2023) trains models to decide which APIs to call and how to integrate responses, establishing tool use as a first-class capability. Practical systems adopt frameworks like LangChain (Mavroudis, 2024), which provide abstractions for agents, tools, memory, and control flow, facilitating composition and observability for real-world deployments. These tools including Python execution and web search cannot run on GPUs like traditional AI workloads and therefore rely on CPU processing.

2.2 CPU Parallelism

Modern CPUs rely on parallelism to improve performance, especially since the end of significant clock speed gains forced a shift to multi-core processors around the mid-2000s (Parkhurst et al., 2006). Multi-threading and multi-processing are foundational parallelism techniques that leverage multi-cores to improve performance through concurrent execution. In a shared-memory model, multi-threading spawns multiple threads within a process that run in parallel on separate cores, all accessing the same

memory space. Alternatively, multi-processing uses multiple processes with isolated memory; this approach is often employed in Python to bypass the Global Interpreter Lock (GIL) for CPU-bound workloads. The choice between threads and processes involves trade-offs: threads incur lower creation and switching overhead but require careful synchronization to avoid race conditions, whereas processes have higher overhead yet eliminate the possibility of direct memory interference between tasks. By leveraging these techniques, CPU-intensive portions of agentic AI workloads can achieve significant speedups, better utilizing CPU resources and feeding data to GPUs more efficiently.

3 CHARACTERIZATION

Prior approaches have characterized agentic LLMs from an algorithmic point of view. (Sapkota et al., 2025) distinguishes agentic AI having distributed cognition, persistent memory, and coordinated planning from traditional single AI agents having task specific automation. Contrary to the algorithmic view, we introduce three orthogonal bases of agentic AI classification that directly influence system metrics. First, on the basis of *orchestrator*, we divide agentic AI systems into LLM-orchestrated and host-orchestrated (through Python code). In the LLM-orchestrated agentic AI workloads, LLM controls the end-to-end execution flow. For instance, a typical flow is shown in Figure 1. In the pipeline, the LLM, working as an orchestrator, decides whether to invoke the tool or emit final output. On the other hand, host-orchestrated workloads calls host/python code to determine the next agent (tool/LLM) in the pipeline. Second, on the basis of *agentic path*, we divide agentic AI systems as static-path and dynamic-path systems. Static path agentic systems follow a predetermined path while dynamic path systems determine the path during runtime based on the orchestrator. In other words, the orchestrator has path decision making capability for dynamic path agentic systems. For static path systems, the orchestrator is only responsible for communication between different agents in the pipeline. Third, on the basis of *agentic flow/repetitiveness*, we divide agentic AI systems into single-step and multi-step systems.

3.1 Orchestrator-Based Classification

This dimension characterizes systems based on where the primary orchestration logic resides. LLM-orchestrated systems delegate control flow decisions to the language model itself, leveraging its reasoning capabilities for task decomposition and execution planning. In contrast, CPU-orchestrated systems employ traditional programmatic control structures, with the CPU managing task scheduling, tool invocation, and result aggregation while treating the LLM as a stateless inference engine. Examples are as follows:

LLM-orchestrated: ReAct (Yao et al., 2023), AutoGPT (Yang et al., 2023), BabyAGI (Nakajima), AgentGPT (age),

CAMEL (Li et al., 2023), MetaGPT (Hong et al., 2024)

Host-orchestrated LangChain (Mavroudis, 2024), Semantic Kernel (microsoft), Haystack (Taulli & Deshmukh, 2025), LlamaIndex (lla), DSPy (Khatab et al., 2023)

3.2 Path-based Classification

This dimension distinguishes between predetermined and adaptive execution strategies. Static path agents follow predefined workflows with deterministic tool invocation sequences. Dynamic-path agents adaptively construct execution graphs based on intermediate results, environmental feedback, and emergent task requirements.

Static Path: LangChain (Mavroudis, 2024), (Nakano et al., 2021), Haystack (Deepset-Ai), LlamaIndex (lla)

Dynamic Path: Tree-of-Thoughts, Graph-of-Thoughts, Reflexion (Shinn et al., 2023), LATS (Zhou et al., 2023a)

3.3 Flow/Repetitiveness-based Classification

This taxonomy captures the iterative nature of agent-environment interactions. One-shot agents complete tasks in a single inference pass without environmental feedback. Multi-shot repetitive agents engage in iterative refinement cycles for complex tasks requiring extensive exploration.

Single-step: CoT prompting systems, Zero-shot tool use, Single-turn QA agents, RAG (Lewis et al., 2020)

Multi-step: WebArena (Zhou et al., 2023b), Balrog (Paglieri et al., 2024), AgentBench (Liu et al., 2023)

3.4 Representative Workloads

3.4.1 Workload Overview

We select five agentic AI workloads for profiling analysis as shown in Table 1. We evaluate Toolformer (Schick et al., 2023) on mathematical benchmarks using WolframAlpha calculator (Wolfram—Alpha), SWE-Agent (Yang et al., 2024) on coding benchmarks using file I/O and Python/Bash execution tools, ChemCrow (Bran et al., 2023) on chemistry research benchmarks using literature search (Arxiv and Pubmed (NCBI)) tool, Haystack (Deepset-Ai) on Question Answering (QA) benchmarks using ENNS retrieval tool, LangChain (Mavroudis, 2024) on QA benchmarks using web search and lexical summarization tools. More details about implementation can be found in Appendix A.

We select these agentic AI workloads because they are representative of different categories of agentic systems, applications and tools. Specifically, First, *challenging applications*: they target factual, coding, and scientific tasks as well as live-data queries where standard LLMs underperform. Second, *diverse computational patterns*: these models span a wide range of model sizes, orchestration pat-

Table 1. Representative Agentic AI systems
(Tools/Application selected for profiling are underlined)

Agentic Workload	Basis of characterization			Tools	Application
	Orchestrator	Path	Flow		
Toolformer (Schick et al., 2023)	LLM	Dynamic	Single-step	Wikipedia Search, <u>Calculator API</u> , Machine Translation System, LLM-based QA, Calendar	QA, MLQA, <u>Math</u>
SWE-Agent (Yang et al., 2024)	LLM	Dynamic	Multi-step	<u>File I/O</u> , Bash/Python Execution	<u>SDE</u> , data analysis
Haystack (Deepset-Ai)	Host	Static	Single-step	Web search, <u>Document Retrieval</u>	<u>QA</u>
ChemCrow (Bran et al., 2023)	LLM	Dynamic	Multi-step	<u>Literature Search (Arxiv/Pubmed)</u> , Molecular tools, Chemical Reaction Tools	<u>Chemistry Research Assistant</u>
Langchain (Mavroudis, 2024)	Host	Static	Single-step	<u>Web search</u> , <u>summarizer</u> , Python code generator/interpreter	<u>QA</u> , Math, DevOps, Summarization

terms and tool integration strategies that are representative of broader agentic AI systems. *Third, academically grounded and industry-relevant*: workloads are curated from peer-reviewed research at top AI conferences as well as from widely adopted open-source repositories used in production.

3.4.2 Toolformer

Toolformer teaches language models to use external tools through self-supervised learning (Schick et al., 2023). It combines a GPT-J 6B (Wang & Komatsuzaki, 2021) model with API call insertion, where the model learns to decide when and how to call tools like calculators, QA systems, and search engines. It achieves 40.4% accuracy on ASDiv math problems, outperforming GPT-3 175B model.

3.4.3 SWE-Agent

SWE-Agent integrates LLM-based reasoning with specialized Agent-Computer Interfaces for automated software engineering (Yang et al., 2024). It provides custom commands for code editing, searching, and navigation optimized for LLM comprehension, achieving 12.5% resolution rate on SWE-bench (Jimenez et al., 2023) benchmarks (4× improvement over baselines). The computation pattern of SWE-Agent primarily involves iterative code refinement and specialized interfaces, which appear in agentic workloads such as Devin (Cognition) and Claude code (Anthropic).

3.4.4 Haystack

Haystack (Deepset-Ai) provides a production-ready framework for building RAG pipelines and question-answering systems. It implements directed multigraph architectures with modular components for retrieval (BM25, dense embeddings) and generation, achieving F1=82.91 on SQuAD 2.0 (Rajpurkar et al., 2018) benchmarks. Haystack computation pattern primarily involves pipeline orchestration and hybrid retrieval, which also appear in agentic workloads such as LlamaIndex (Ila) and Semantic Kernel (microsoft).

We choose ENNS retrieval from C4 (Dodge et al., 2021) document corpus (305 GB english variant). In a controlled QA-RAG study (Quinn et al., 2025), ENNS outperform Approximate Nearest Neighbor Search (ANNS) in generation accuracy by 22.6–53.4% at K=1 and 13.6–45.2% at K=16 across FiDT5, Llama-3-8B, and Llama-3-70B models. Moreover, the paper also concluded that ENNS dominates the throughput-accuracy Pareto frontier as compared to ANNS. Following the same setting used in the paper ((Quinn et al., 2025)), we choose CPU-based FAISS (Douze et al., 2024) retrieval due to large document size (> 300 GB), significantly exceeding the GPU memory.

3.4.5 ChemCrow

ChemCrow (Bran et al., 2023) augments LLMs with specialized chemistry tools for scientific research automation. ChemCrow integrates 18 expert-designed tools spanning reaction prediction, molecular analysis, and safety assessment, using ReAct-style reasoning chains. It outperforms GPT-4 by 4.4/10 points in expert evaluations and achieves 100% success rate on synthesis tasks. The computation pattern of ChemCrow primarily involves domain-specific tool integration and ReAct reasoning, which also appear in other agentic workloads such as GeoGPT (Zhang et al., 2023).

3.4.6 LangChain

LangChain (Mavroudis, 2024) facilitates composable agent development through modular chains and graph-based orchestration. LangChain consists of core abstractions for tool calling, memory management, and stateful multi-agent coordination. The computation pattern of LangChain primarily involves chain composition and stateful orchestration, which also appear in other agentic workloads such as CrewAI (CrewAI, 2025) and AutoGen (Wu et al., 2024).

We choose a custom agentic pipeline (web search – > summarization – > LLM inference) inspired by the web search feature of popular chatbots including ChatGPT. The

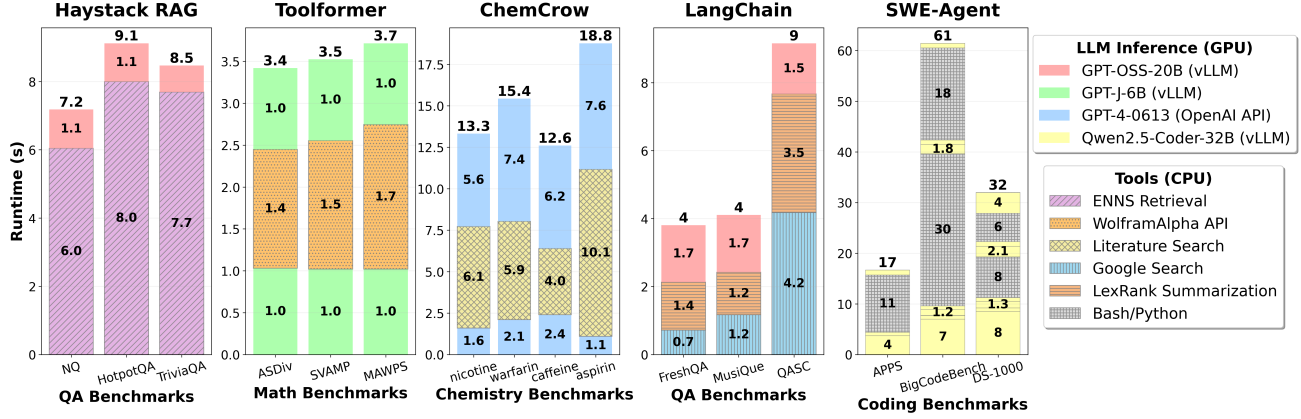


Figure 2. (a) Haystack with ENNS retrieval on QA benchmarks (b) Toolformer with WolframAlpha API on Math benchmarks (c) ChemCrow with literature (Arxiv/Pubmed) search tool on Chemistry benchmarks (d) LangChain with web search and LexRank summarization tools on QA benchmarks (e) Mini-SWE-Agent with bash/Python execution tools on coding benchmarks

role of summarizer is to reduce the prompt length and parse factual information from the web documents. We chose a CPU-based LexRank summarizer (Erkan & Radev, 2004) compared to an LLM-based summarizer because of three reasons. *First, Hallucinations:* A study (Maynez et al., 2020) shows that on the XSum benchmark (Narayan et al., 2018), 73–79% of model summaries contained at least one hallucination; the best system still had 64% extrinsic hallucinations and only 27–35% of outputs were judged factual overall. *Second, Domain Accuracy:* The accuracy of LexRank-based summarizer is within 0.05 ROUGE-1 of LLM-based summarizer for DUC-2004 benchmark and even surpasses for legal benchmark, such as BillSum (Giarelis et al., 2023). *Third, Cost-efficient:* Given that GPU is a costly resource, choosing CPU for summarization makes the customized agentic pipeline more efficient.

4 PROFILING AND KEY TAKEAWAYS

4.1 Profiling Setup

The experiments are performed on a state-of-the-art (SOTA) system with 48-core (2 threads each) Intel Emerald Rapids CPU (DDR5 DRAM) and NVIDIA B200 GPU (HBM3e).

4.2 Latency

Figure 2 profiles end-to-end runtime for five representative agentic AI workloads—Haystack RAG, Toolformer, ChemCrow, LangChain, and Mini-SWE-Agent—across QA, math, chemistry, QA, and SWE benchmarks respectively. Across all settings, the dominant contribution to latency is mostly tool processing (retrieval, WolframAlpha API, literature search, LexRank summarization, and Bash/Python execution) running on CPU, not LLM inference.

For QA with Haystack RAG, retrieval is the main bottleneck

consuming 6.0 s for NQ (Kwiatkowski et al., 2019), 8.0 s for HotpotQA (Yang et al., 2018), and 7.7 s for TriviaQA (Joshi et al., 2017), i.e., 84.5–90.6% of runtime, with LLM inference contributing ≤ 0.5 s. On math benchmarks with Toolformer, the first GPT-J 6B model inference is constant (1 s) due to the benchmarks having similar number of input tokens, while WolframAlpha API calls add 1.4–1.7 s depending on the benchmark. The final inference is also constant as the tool output only adds a few token to the prompt used during the first inference. ChemCrow shows large tool latencies where the literature search accounts for 4.0–10.1 s including arxiv paper search, download and read, as well as Pubmed paper search and abstract read. The final GPT-4-0613 inference accounts for 5.6–7.6 s, pushing totals to 12.6–18.8 s depending on the benchmark. For QA benchmarks with LangChain, web search (up to 4.2 s) or summarization tool (up to 3.5 s) can drive more than half of the end-to-end latency. The pattern reinforces that constraining the number of websites to web search and summarize is the primary optimization scheme, not the choice of base model. For SWE-Agent, Bash/Python execution account for 64.7%, 78.7%, and 43.8% of the total latency for APPS (Hendrycks et al., 2021), BigCodeBench (Zhuo et al., 2024), and DS-1000 (Lai et al., 2023) benchmarks respectively.

Key Takeaway 1: Tool processing on CPUs can significantly impact (up to 90.6%) the execution latency for agentic workloads, motivating a joint CPU-CPU optimization instead of GPU only.

4.3 Throughput

We begin by analyzing CPU parallelism in Section 4.3.1 and deriving effective strategies. Thereafter, we pair the learnt strategy on CPU side along with well-studied GPU parallelization strategy to parallelize multiple agentic requests.

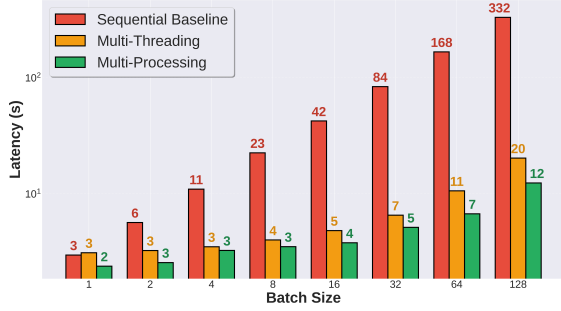


Figure 3. Comparison of multi-processing and multi-threading with sequential baseline (single core) for Langchain workload

However, we identify two throughput bottlenecks caused by either GPU factors (limited by device-memory capacity and bandwidth) or CPU factors (limited by coherence, synchronization, core over-subscription, and host-memory capacity/bandwidth). We examine these bottlenecks in [Section 4.3.2](#) and [Section 4.3.3](#) respectively. Finally, we show the throughput saturation through the representative workloads caused by CPU and GPU factors in [Section 4.3.4](#).

4.3.1 CPU Parallelism for Agentic Workloads

We compare the effect of multi-processing and multi-threading with the sequential baseline (single core) for Langchain workload in [Figure 3](#). LangChain’s built-in `Runnable.batch` API processes multiple inputs concurrently within one process implemented via a thread pool (multi-threading). Multi-processing launches N independent Python processes (each serving batch size 1) using the shell background operator ‘&’, thereby achieving coarse-grained parallelism across CPU cores and sidestepping single-process Global Interpreter Lock (GIL) limitations. Moreover, multi-processing also mitigates the synchronization overheads incurred by multi-threading. The performance is nearly similar for low batch size. However, for batch size 128, multi-processing achieves $26.8\times$ and $1.6\times$ speedup against baseline and multi-threading respectively.

For workloads including haystack using significant memory (~ 300 GB) during retrieval, multi-processing is highly ineffective due to independent memory usage. Therefore, we choose multi-threading for haystack workload to parallelize multiple retrieval tasks effectively using shared memory. However, for rest of the workloads, we use multi-processing for parallelizing multiple tasks for better performance.

4.3.2 GPU Throughput Bottlenecks

[Figure 4a](#) shows the throughput- $(input + output\ tokens)/s$ variation as we scale the batch size from 1 to 128 for different input and output token lengths. The local vLLM server delivers large throughput through PagedAttention ([Kwon](#)

[et al., 2023](#)). Across all token configurations, throughput scales nearly linearly with batch size up to 64 requests, after which additional batching yields diminishing returns and the gain plateaus. The saturation is consistent with memory-bound behavior: as batch size grows, the key-value (KV) cache expands proportionally to the total tokens and begins to exceed GPU high-bandwidth memory, even under PagedAttention, which minimizes fragmentation but cannot eliminate capacity pressure. This result resonates with a recent work ([Recasens et al., 2025](#)) which states that large-batch inference remains memory-bound with main memory bandwidth saturation as the primary bottleneck. Even without spilling, a large KV cache usage can saturate the throughput. For instance, the paper reports that OPT-1.3B achieves almost maximum throughput using just 40% of its KV cache, while OPT-2.7B requires 50%. Increasing batch size further yields only marginal throughput gains, at the cost of a larger GPU memory usage.

Moreover, a large KV cache forces paging or spilling between GPU and host memory. The resulting data movement is gated by the substantially lower bandwidth of PCIe relative to on-device memory, leading to transfer-induced stalls that cap end-to-end throughput. Offloading-based LLM serving systems repeatedly identify PCIe as the dominant bottleneck once KV caches no longer fit in the main memory. FlexGen ([Sheng et al., 2023](#)) shows that for OPT-175B, the total memory required to store the KV cache is 1.2 TB, which is $3.8\times$ the model weights, identifying the KV cache as a key bottleneck in large-batch inference.

4.3.3 CPU Throughput Bottlenecks

CPU throughput on multi-core systems can saturate well before all cores are busy. For instance, A dual-socket Haswell node reaches $>80\%$ of peak bandwidth on the STREAM benchmark with only four processes per socket, so more cores deliver diminishing returns ([Balay et al., 2019](#)). Even when aggregate cores increase, node-to-node bandwidth and latency constrain scaling: in another STREAM study ([Bergstrom, 2011](#)), an Intel 4-socket system saturated near 32 cores and peaked around 40 GB/s, whereas a NUMA-aware AMD machine continued scaling to 48 cores and ~ 55 GB/s. Placement can change performance by up to $2.37\times$, with memory-access latency differences of hundreds to thousands of cycles across inter-node topologies, underscoring the role of interconnect bandwidth and coherence traffic. As working sets exceed private caches, cache-line ping-pong and false sharing under MESI increase coherence traffic, and remote memory references incur higher latency that stall pipelines and saturate on-socket fabrics. ([Mattson et al., 2008](#)) argues that the overhead of cache coherence restricts the ability to scale up to even 80 cores. In parallel, synchronization hot spots including global barriers, contended locks, and atomic operations add queuing and serialization

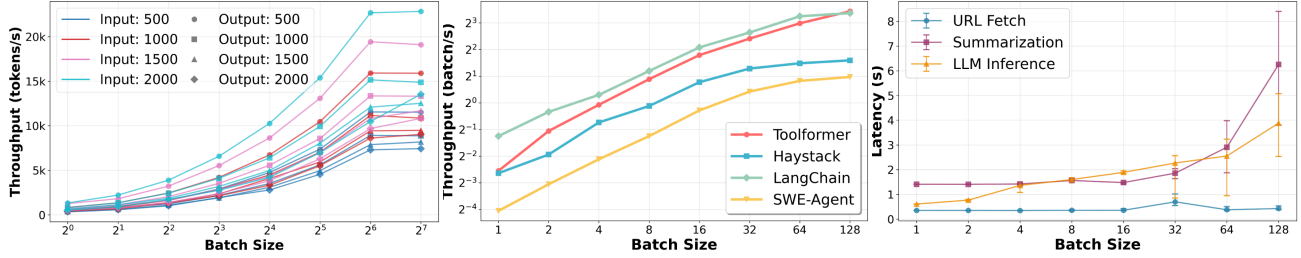


Figure 4. (a) vLLM throughput saturation for GPT-OSS-20B model (b) Throughput saturation for various agentic workloads (c) Average time taken by different components in Langchain benchmark showing a critical CPU context switching bottleneck at batch size 128

delays, where barrier time is dictated by the slowest thread called “straggler”, and tail latency rises sharply with contention. If we increase the number of parallel processes beyond the available cores (over-subscription), OS scheduler contention, context switching and lost of hardware state (caches/TLB) overheads dominate (Iancu et al., 2010).

4.3.4 Throughput Saturation of Agentic Workloads

In the previous two subsections, we described that the throughput of agentic workloads is either saturated by GPU or CPU parallelization bottlenecks at large batch sizes. In Figure 4b, we plot the throughput variation of different agentic workloads with batch size scaling. We parallelize each component of the agentic workload including API calls, LLM inference and tool processing on CPU. We showcase different scenarios of throughput boundedness through Toolformer, Haystack RAG, Langchain and SWE-Agent workloads on MAWPS, NQ, FreshQA and APPS benchmarks respectively. For Toolformer workload, we see the rate of throughput increase keeps slowing down from $2.8\times$ at batch size 2 to $1.4\times$ at batch size 128. The WolframAlpha API calls are parallelized with nearly zero latency overheads but KV cache size keeps increasing as we increase the batch size. As we increase the batch size beyond 128, we will see the throughput saturation as observed in Figure 4a. For Haystack RAG workload, retrieval is bottle-necked beyond batch size 32 due to LLC pressure and disk I/O contention arising out of the huge size of the C4 documents. For Langchain and SWE-Agent workloads, the throughput saturates at batch size 128 due to core over-subscription. Figure 4c further shows that the impact of over-subscription in Langchain workload where the average latency of summarization task increases from 2.9 s at batch size 64 to 6.3 s at batch size 128. The saturation also comes partly from GPU memory bottleneck where the average latency of LLM inference increases from 2.6 s at batch size 64 to 3.9 s at batch size 128. The URL fetch stage is parallelized with nearly zero latency overheads just like WolframAlpha API calls used in Toolformer workload.

Key Takeaway 2: The throughput of agentic AI workloads is either saturated by CPU factors (core over-subscription, cache-coherence, synchronization) or GPU factors (device-memory capacity, bandwidth).

4.4 Energy

Owing to facility constraints, energy was measured on a separate host equipped with an AMD Ryzen Threadripper PRO 7985WX (64 cores) and NVIDIA H200 GPU. CPU energy was obtained via *pyRAPL*, which reads on-chip RAPL (David et al., 2010) energy counters. GPU energy was computed by numerically integrating board power reported by *nvidia-smi* sampled every 100 ms using the composite trapezoidal rule ((Burden et al., 2010)) over time. In the quiescent (idle) state, the CPU platform drew 113 W, and the GPU board drew 115 W. The idle state power was subtracted from the measured power to get dynamic power.

As shown in Figure 5, the dynamic energy consumption profile of Langchain workload on FreshQA benchmark demonstrates a stark non-linear scaling pattern as batch size increases, with total dynamic energy rising from 108 joules at batch size 1 to 4114 joules at batch size 128, representing a 38.1 -fold increase despite the 128 -fold increase in batch size. While GPU dynamic energy scales from 86 to 2307 joules ($26.8\times$ increase), CPU energy increases substantially from 22 joules at batch size 1 to 1807 joules at batch size 128 ($86.7\times$ increase). The results show that GPU parallelism is fundamentally more energy efficient as compared to CPU multiprocessing. This disproportionate CPU energy scaling (20% at small batch sizes to 44% at batch size 128) fundamentally shifts the system’s energy distribution. While measured on different hardware, the relative trends in CPU vs GPU energy consumption remain architecturally consistent across modern server-class systems.

Key Takeaway 3: CPU dynamic energy share becomes significant (44%) at large batch size (128), as CPU parallelism is less energy efficient compared to GPU.

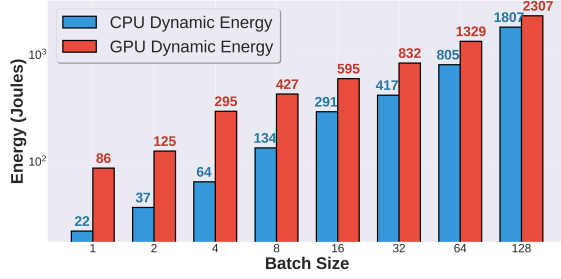


Figure 5. CPU (AMD Threadripper) and GPU (Nvidia B200) dynamic energy consumption for Langchain workload

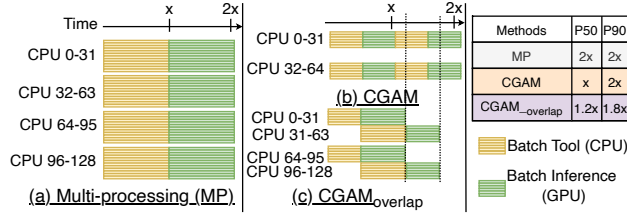


Figure 6. Timeline of batched agentic AI inference for (a) Multi-processing, (b) CGAM, and (c) CGAM_{overlap}

5 OPTIMIZATIONS

Based on throughput saturation insights (Section 4.3), we present two scheduling optimizations- ① CPU and GPU Aware Micro-batching (CGAM- Section 5.1) and ② Mixed Agentic Workload Scheduling (MAWS- Section 5.2) for homogeneous and heterogeneous workloads respectively.

5.1 CGAM

As seen in Figure 4b, throughput exhibits saturation when gains diminish as batch size increases. The saturation arises due to the sharp rise in median and tail latency at large batch size (128) as seen in Figure 4b, where both the median and tail latencies of CPU-bound summarization stage increased by more than $2\times$. We introduce CGAM to cap the batch size and use micro-batching to optimize both median and tail latencies for large batch agentic AI inference.

5.1.1 Batching Cap Selection

Let $T(B)$ denote the throughput (requests/second) for batch size B . We define the throughput gain ratio as $r(B) = \frac{T(B)}{T(B/2)}$ which captures the speedup achieved by doubling the batch size. The optimal batching cap B_{cap} should maximize resource efficiency while avoiding the saturation regime where additional parallelism yields negligible improvements. We seek the smallest batch size beyond which the throughput gain ratio falls below a threshold:

$$B_{cap} = \max\{B \in \{2^k : k \in N\} : r(B) > \lambda\} \quad (1)$$

Table 2. Throughput gain ratios r and selected B_{cap} values

Workload	$r(64)$	$r(128)$	B_{cap}
Langchain	1.52	1.09	64
Haystack	1.15	1.08	64
SWE-Agent	1.32	1.10	64

where λ represents the acceptable efficiency threshold. From our experimental analysis, setting $\lambda = 1.1$ provides a practical balance, meaning we stop increasing batch size when doubling it yields $< 10\%$ throughput improvement.

Table 2 applies this selection criterion against our profiled workloads, demonstrating that choosing $B_{cap} = 64$ aligns with the point where throughput gains fall below λ .

CGAM processes each micro-batch sequentially with maximum parallelism of B_{cap} as shown in Figure 6 for $B = 128$ and $B_{cap} = 64$. The workload execution pattern is Tools (CPU) \rightarrow LLM inference (GPU) inspired by Langchain and Haystack workloads.

5.1.2 Advantages of CGAM

CGAM provides three advantages compared to multi-processing. *First*, $\sim 2\times$ improvement in P50: Assuming nearly equal end-to-end latency due to saturation, the first micro-batch will finish around half of the total latency (Figure 6). This is beneficial in cases of tiered serving system where different users are tiered differently based on amount of money they spend. Using CGAM, the top 50% tier of users can get $\sim 2\times$ better service while maintaining the same service for the bottom 50% tier of users compared to the baseline. *Second*, $\sim 0.5\times$ KV cache usage: At any time, we are running half of the total batches, reducing KV cache usage by almost half. This is beneficial in cases where we have limited GPU memory available and have to rely on slow PCIe communication due to CPU offloading of the KV cache. *Third*, $\sim 2\times$ CPU energy reduction: Limiting the number of cores through B_{cap} , we can also save $\frac{Max\ cores}{B_{cap}} = \frac{128}{64} = 2\times$ energy assuming each cores consumes equal power and equal end-to-end latency. As shown in Section 4.4, CPU energy is a dominant factor of the total energy consumption at large batch sizes. Therefore, reduction in CPU energy reduces a major portion of the total energy, making CGAM highly energy-efficient.

5.1.3 CGAM_{overlap}

We can also utilize the remaining idle CPUs for more speed-up at the cost of energy. For mixed agentic workloads with comparable CPU and GPU latencies, we present CGAM_{overlap} to overlap the two micro-batches as shown in Figure 6. Instead of waiting for first micro-batch to finish completely, we can start executing the second micro-batch

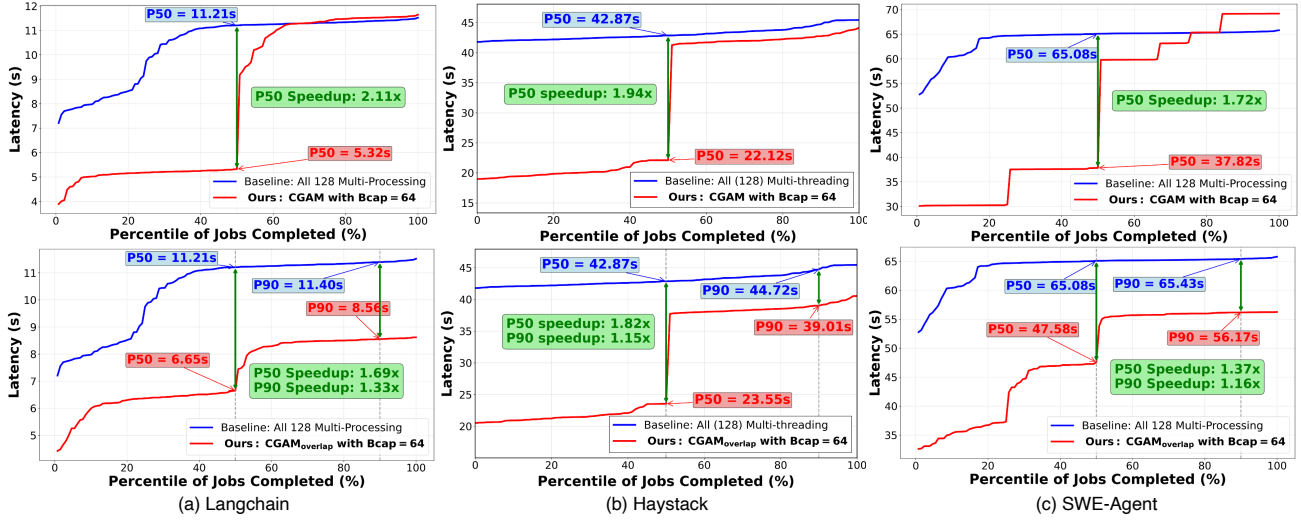


Figure 7. Comparison of CGAM and $CGAM_{overlap}$ using $Bcap = 64$ against baseline (multi-processing or multi-threading) on (a) Langchain workload on FreshQA benchmark, (b) Haystack workload on NQ benchmark and (c) SWE-Agent on APPS benchmark

when CPU portion of the first micro-batch finishes. At that time, we will run GPU portion of the first micro-batch and CPU portion of the second micro-batch concurrently. The higher CPU contention during the concurrent execution increases P50 latency compared to CGAM. However, overlapping decreases P90 latency as the second micro-batch starts much earlier as compared to CGAM as shown in Figure 6.

5.2 MAWS

Agentic workloads can be heterogenous: some are CPU-heavy (dominated by tool executions on CPUs), while others are LLM-heavy (dominated by inference time on GPU). This paper so far has focused on CPU-heavy requests. However, there are agentic workloads with LLM heavy components as well. For example, some ChatGpt requests consist of web search component, while some requests are purely LLM inference with minimal use of tools.

We chose multi-processing for parallelizing the CPU-bound Langchain workload (Section 4.3.1). However, this becomes ineffective for heterogeneous agentic workloads. Suppose we have two different types of agentic workloads - CPU heavy and LLM heavy. The LLM heavy tasks would cause over-subscription of the CPU resources during multi-processing, making the CPU-heavy tasks less effective. Therefore, we need to limit the CPU usage of LLM heavy tasks. Since, they are LLM-heavy, we can use the lighter multi-threading for parallel vLLM API I/O. This frees up a lot of CPU resources making the CPU heavy tasks more effective. Therefore, we can optimize mixed agentic AI inference through adaptive multi-processing and multi-threading for CPU-heavy and LLM-heavy workloads respectively.

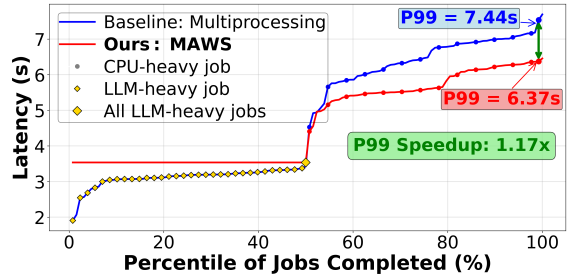


Figure 8. Comparison of MAWS against multiprocessing baseline on 128 mixed Langchain tasks (half LLM heavy, half CPU heavy)

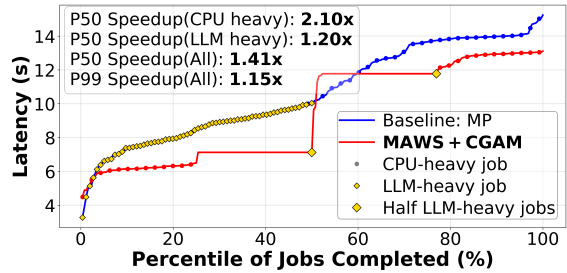


Figure 9. Comparison of MAWS+CGAM against multiprocessing baseline on 256 mixed Langchain tasks

6 EVALUATION

6.1 Methodology

We assume a closed-loop arrival system (all B requests arrive simultaneously at $t=0$) to evaluate CGAM (Section 6.2) and MAWS (Section 6.3) for $B = 128$ and combined MAWS+CGAM (Section 6.4) for $B = 256$. We use the

same system setup used in profiling (Intel Emerald Rapids CPU and B200 GPU). For CGAM, We evaluate on Haystack RAG, Langchain and SWE-Agent workloads having significant tool processing components (ENNS retrieval, LexRank summarization and Bash/Python execution). Other representative workloads (Toolformer and ChemCrow) involve a lot of external processing due to WolframAlpha API and OpenAI API. For baseline, we apply the same approach described in Section 4.3.1. We use multi-processing for Langchain and SWE-Agent workloads due to CPU-bound tool processing and multi-threading for Haystack RAG workload due to the large size of C4 documents. For MAWS, we use Langchain workload because it can be customized to create custom heterogeneous agentic pipelines whereas other workloads have fixed pipeline. All the evaluations show single run times. Statistical variance of 5% is observed during multiple runs and therefore has a low impact on P50 latency speedups.

6.2 CGAM

Figure 7 quantifies the effect of CGAM and $\text{CGAM}_{\text{overlap}}$ using $B_{\text{cap}} = 64$ on LangChain, Haystack and SWE-Agent workloads. CGAM results in $2.11\times$ (11.21 s to 5.32 s), $1.94\times$ (42.87 s to 22.12 s) and $1.72\times$ (65.08 s to 37.82 s) respective reduction in P50 latency using $\frac{3}{4}^{\text{th}}$ of the total CPUs (96) while maintaining nearly same tail latency. Moreover, assuming equal power consumption throughout the execution, CGAM saves $\sim 1.5\times$ CPU dynamic energy. As per Figure 6, we see the expected $\sim 2\times$ reduction in P50 latency. Some variance from $2\times$ is observed due to uneven CPU-GPU execution ratios and different throughput saturation impacts on CPU and GPU components.

$\text{CGAM}_{\text{overlap}}$ results in 1.69, 1.82, and $1.37\times$ reduction in P50 latency as well as 1.33, 1.15, and $1.16\times$ reduction in P90 latency respectively for LangChain, Haystack and SWE-Agent workloads. As per Figure 6, we observe the expected trade off for reduced P90 latency at the cost of higher P50 latency. We observe higher P90 reduction in Langchain workload because overlap favors the relatively equal CPU and GPU execution latencies of Langchain compared to Haystack and SWE-Agent where CPU latencies dominate.

6.3 MAWS

We use two different Langchain pipelines for heterogeneous workloads. We use the same pipeline used in profiling to represent CPU-heavy workload, while we use a different pipeline (guardrail \rightarrow LLM inference) to create LLM heavy workload, where the guardrail is a simple *if-else* function to check for malicious prompts. Figure 8 shows the comparison between MAWS against multi-processing baseline for an equal mix of both the Langchain pipelines for a total of 128 tasks. MAWS performs $1.17\times$ better in

terms of P99 latency while maintaining similar P50 latency.

6.4 MAWS+CGAM

Figure 9 shows the evaluation of both MAWS+CGAM together on the same workload mix used in Figure 8 for a total of 256 tasks so that we achieve throughput saturation for half (128) of the tasks which are CPU-heavy. MAWS+CGAM performs $2.1\times$, $1.2\times$, and $1.4\times$ better than baseline in terms of P50 latency for CPU-heavy task, LLM-heavy tasks and all tasks respectively. Furthermore, MAWS+CGAM saves $1.15\times$ on overall P99 latency.

7 RELATED WORKS

Agentic AI Characterization: A recent work (Sapkota et al., 2025) distinguishes agentic AI with distributed cognition, persistent memory, and coordinated planning from traditional single AI agents with task specific automation. Contrary to this algorithmic view, we characterize agentic AI from a systems point of view.

Agentic AI Profiling: A recent work (Kim et al., 2025) profiled some agentic workloads focused on reasoning from a GPU-centric perspective without exposing the CPU bottleneck due to tool processing. Most of the tools they used are API calls (WolframAlpha and Wikipedia) and can easily be parallelized. Another work (Asgar et al., 2025) profiled agentic AI workloads and optimized the orchestration framework but focused solely on external tool calls. Therefore, the work was based on nearly zero local CPU overhead, lacking a comprehensive CPU-centric perspective.

Scheduling Optimizations: A lot of scheduling works in literature focused solely on LLM inference optimization. A recent work (Recasens et al., 2025) introduced micro-batching optimization to alleviate the memory bandwidth bottleneck during the decode phase. However, the paper did not consider CPU micro-batching. Orca (Yu et al., 2022) and vLLM (Vellaisamy et al., 2025) use continuous batching on first come first serve (FCFS) basis. Our approach orthogonally use continuous batching for LLM inference with (i) CPU-Aware batching cap selection, (ii) CPU/GPU overlapping and (iii) adaptive multi-processing/multi-threading which are not present in prior works.

8 CONCLUSION

This work brings forth a CPU-centric perspective on agentic AI. We systematically characterize Agentic AI on the basis of orchestrator, inference path dynamics and repetitiveness of the agentic flow which directly affect the system-level performance. We choose five representative workloads from the characterization and profile latency, throughput and energy to demystify the CPU bottlenecks. We also introduce

two key optimizations - CGAM and MAWS for homogeneous and heterogeneous agentic workloads respectively. Our profiling and results are demonstrated on state-of-the-art Intel Emerald Rapids CPU and NVIDIA B200 GPU system; studies on diverse hardware configurations remain future work.

ACKNOWLEDGEMENTS

We thank Zishen Wan, Akshat Ramachandran, and Sarbartha Banerjee for valuable feedback that helped improve this work.

REFERENCES

- AgentGPT. <https://agentgpt.reworkd.ai/>.
- LlamaIndex - Build Knowledge Assistants over your Enterprise Data. <https://www.llamaindex.ai/>.
- Abdin, M., Aneja, J., Behl, H., Bubeck, S., Eldan, R., Gunasekar, S., Harrison, M., Hewett, R. J., Javaheripi, M., Kauffmann, P., et al. Phi-4 technical report. *arXiv preprint arXiv:2412.08905*, 2024.
- Anthropic. Claude code. <https://www.claude.com/product/claude-code>.
- Asgar, Z., Nguyen, M., and Katti, S. Efficient and scalable agentic ai with heterogeneous systems. *arXiv preprint arXiv:2507.19635*, 2025.
- Balay, S., Abhyankar, S., Adams, M., Brown, J., Brune, P., Buschelman, K., Dalcin, L., Dener, A., Eijkhout, V., Gropp, W., et al. Petsc users manual. 2019.
- Bedi, S., Liu, Y., Orr-Ewing, L., Dash, D., Koyejo, S., Callahan, A., Fries, J. A., Wornow, M., Swaminathan, A., Lehmann, L. S., et al. Testing and evaluation of health care applications of large language models: a systematic review. *Jama*, 2025.
- Belcak, P., Heinrich, G., Diao, S., Fu, Y., Dong, X., Muralidharan, S., Lin, Y. C., and Molchanov, P. Small language models are the future of agentic ai, 2025. URL <https://arxiv.org/abs/2506.02153>.
- Berglund, L., Stickland, A. C., Balesni, M., Kaufmann, M., Tong, M., Korbak, T., Kokotajlo, D., and Evans, O. Taken out of context: On measuring situational awareness in llms. *arXiv preprint arXiv:2309.00667*, 2023.
- Bergstrom, L. Measuring numa effects with the stream benchmark. *arXiv preprint arXiv:1103.3225*, 2011.
- Bran, A. M., Cox, S., Schilter, O., Baldassari, C., White, A. D., and Schwaller, P. Chemcrow: Augmenting large-language models with chemistry tools. 2023.
- Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.
- Burden, R., Faires, J., and Burden, A. Numerical analysis, cengage learning. *Brooks/Cole*, 2010.
- Cognition. Devin: The ai software engineer. <https://devin.ai/>.
- CrewAI. Fast and flexible multi-agent automation framework, 2025. URL <https://github.com/crewAIInc/crewAI>. Open-source multi-agent orchestration framework for collaborative AI agents.
- Cui, Y., Huang, S., Zhong, J., Liu, Z., Wang, Y., Sun, C., Li, B., Wang, X., and Khajepour, A. Drivellm: Charting the path toward full autonomous driving with large language models. *IEEE Transactions on Intelligent Vehicles*, 9(1): 1450–1464, 2023.
- David, H., Gorbato, E., Hanebutte, U. R., Khanna, R., and Le, C. Rapl: Memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, pp. 189–194, 2010.
- Deepset-Ai. haystack. <https://github.com/deepset-ai/haystack>.
- Dodge, J., Sap, M., Marasović, A., Agnew, W., Ilharco, G., Groeneveld, D., Mitchell, M., and Gardner, M. Documenting large webtext corpora: A case study on the colossal clean crawled corpus. *arXiv preprint arXiv:2104.08758*, 2021.
- Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P.-E., Lomeli, M., Hosseini, L., and Jégou, H. The faiss library. *arXiv preprint arXiv:2401.08281*, 2024.
- Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al. The llama 3 herd of models. *arXiv e-prints*, pp. arXiv–2407, 2024.
- Erkan, G. and Radev, D. R. Lexrank: Graph-based lexical centrality as salience in text summarization. *Journal of artificial intelligence research*, 22:457–479, 2004.
- Fu, D., Li, X., Wen, L., Dou, M., Cai, P., Shi, B., and Qiao, Y. Drive like a human: Rethinking autonomous driving with large language models. In *2024 IEEE/CVF Winter Conference on Applications of Computer Vision Workshops (WACVW)*, pp. 910–919. IEEE, 2024.

- Gan, W., Qi, Z., Wu, J., and Lin, J. C.-W. Large language models in education: Vision and opportunities. In *2023 IEEE international conference on big data (BigData)*, pp. 4776–4785. IEEE, 2023.
- Giarelis, N., Mastrokostas, C., and Karacapilidis, N. Abstractive vs. extractive summarization: An experimental review. *Applied Sciences*, 13(13):7620, 2023.
- Gunasekar, S., Zhang, Y., Aneja, J., Mendes, C. C. T., Del Giorno, A., Gopi, S., Javaheripi, M., Kauffmann, P., de Rosa, G., Saarikivi, O., et al. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*, 2023.
- Hendrycks, D., Burns, C., Basart, S., Zou, A., Mazeika, M., Song, D., and Steinhardt, J. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.
- Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.
- Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Zhang, C., Wang, J., Wang, Z., Yau, S. K. S., Lin, Z., et al. Metagpt: Meta programming for a multi-agent collaborative framework. *International Conference on Learning Representations, ICLR*, 2024.
- Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Lu, K., et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.
- Iancu, C., Hofmeyr, S., Blagojević, F., and Zheng, Y. Over-subscription on multicore processors. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1–11. IEEE, 2010.
- Jablonka, K. M., Schwaller, P., Ortega-Guerrero, A., and Smit, B. Leveraging large language models for predictive chemistry. *Nature Machine Intelligence*, 6(2):161–169, 2024.
- Jiang, W., Subramanian, S., Graves, C., Alonso, G., Yazdankhsh, A., and Dadu, V. Rago: Systematic performance optimization for retrieval-augmented generation serving. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, pp. 974–989, 2025.
- Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.
- Joshi, M., Choi, E., Weld, D. S., and Zettlemoyer, L. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. *arXiv preprint arXiv:1705.03551*, 2017.
- Khattab, O., Singhvi, A., Maheshwari, P., Zhang, Z., Santhanam, K., Vardhamanan, S., Haq, S., Sharma, A., Joshi, T. T., Moazam, H., et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.
- Khot, T., Clark, P., Guerquin, M., Jansen, P., and Sabharwal, A. Qasc: A dataset for question answering via sentence composition. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 8082–8090, 2020.
- Kim, J., Shin, B., Chung, J., and Rhu, M. The cost of dynamic reasoning: Demystifying ai agents and test-time scaling from an ai infrastructure perspective. *arXiv preprint arXiv:2506.04301*, 2025.
- Komeili, M., Shuster, K., and Weston, J. Internet-augmented dialogue generation. *arXiv preprint arXiv:2107.07566*, 2021.
- Koncel-Kedziorski, R., Roy, S., Amini, A., Kushman, N., and Hajishirzi, H. Mawps: A math word problem repository. In *Proceedings of the 2016 conference of the north american chapter of the association for computational linguistics: human language technologies*, pp. 1152–1157, 2016.
- Kwiatkowski, T., Palomaki, J., Redfield, O., Collins, M., Parikh, A., Alberti, C., Epstein, D., Polosukhin, I., Devlin, J., Lee, K., et al. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:453–466, 2019.
- Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.
- Lai, Y., Li, C., Wang, Y., Zhang, T., Zhong, R., Zettlemoyer, L., Yih, W.-t., Fried, D., Wang, S., and Yu, T. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pp. 18319–18345. PMLR, 2023.
- Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.
- Li, G., Hammoud, H., Itani, H., Khizbullin, D., and Ghanem, B. Camel: Communicative agents for” mind” exploration of large language model society. *Advances in Neural Information Processing Systems*, 36:51991–52008, 2023.

- Lin, S., Hilton, J., and Evans, O. Truthfulqa: Measuring how models mimic human falsehoods. *arXiv preprint arXiv:2109.07958*, 2021.
- Liu, X., Yu, H., Zhang, H., Xu, Y., Lei, X., Lai, H., Gu, Y., Ding, H., Men, K., Yang, K., et al. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023.
- Mattson, T. G., Van der Wijngaart, R., and Frumkin, M. Programming the intel 80-core network-on-a-chip terascale processor. In *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pp. 1–11. IEEE, 2008.
- Mavroudis, V. Langchain. 2024.
- Maynez, J., Narayan, S., Bohnet, B., and McDonald, R. On faithfulness and factuality in abstractive summarization. *arXiv preprint arXiv:2005.00661*, 2020.
- Miao, S.-Y., Liang, C.-C., and Su, K.-Y. A diverse corpus for evaluating and developing english math word problem solvers. *arXiv preprint arXiv:2106.15772*, 2021.
- microsoft. GitHub - microsoft/semantic-kernel: Integrate cutting-edge LLM technology quickly and easily into your apps. <https://github.com/microsoft/semantic-kernel>.
- Nakajima, Y. Babyagi, 2023. <https://github.com/yoheinakajima/babyagi>.
- Nakano, R., Hilton, J., Balaji, S., Wu, J., Ouyang, L., Kim, C., Hesse, C., Jain, S., Kosaraju, V., Saunders, W., et al. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.
- Narayan, S., Cohen, S. B., and Lapata, M. Don't give me the details, just the summary! topic-aware convolutional neural networks for extreme summarization. *arXiv preprint arXiv:1808.08745*, 2018.
- NCBI. Pubmed. URL <https://pubmed.ncbi.nlm.nih.gov/>.
- Ouyang, Q., Wang, S., and Wang, B. Enhancing accuracy in large language models through dynamic real-time information injection. 2023.
- Paglieri, D., Cupiał, B., Coward, S., Piterbarg, U., Wolczyk, M., Khan, A., Pignatelli, E., Kuciński, Ł., Pinto, L., Fergus, R., et al. Balrog: Benchmarking agentic llm and vlm reasoning on games. *arXiv preprint arXiv:2411.13543*, 2024.
- Parkhurst, J., Darringer, J., and Grundmann, B. From single core to multi-core: preparing for a new exponential. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pp. 67–72, 2006.
- Patel, A., Bhattamishra, S., and Goyal, N. Are nlp models really able to solve simple math word problems? *arXiv preprint arXiv:2103.07191*, 2021.
- Patel, A., Hofmarcher, M., Leoveanu-Condrei, C., Dinu, M.-C., Callison-Burch, C., and Hochreiter, S. Large language models can self-improve at web agent tasks. *arXiv preprint arXiv:2405.20309*, 2024.
- Princeton NLP Group. Swe-llama-7b. <https://huggingface.co/princeton-nlp/SWE-Llama-7b>, 2023. Hugging Face model card; accessed 2025-10-05.
- Quinn, D., Nouri, M., Patel, N., Salihu, J., Salemi, A., Lee, S., Zamani, H., and Alian, M. Accelerating retrieval-augmented generation. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pp. 15–32, 2025.
- Rajpurkar, P., Jia, R., and Liang, P. Know what you don't know: Unanswerable questions for squad. *arXiv preprint arXiv:1806.03822*, 2018.
- Recasens, P. G., Agullo, F., Zhu, Y., Wang, C., Lee, E. K., Tardieu, O., Torres, J., and Berral, J. L. Mind the memory gap: Unveiling gpu bottlenecks in large-batch llm inference. *arXiv preprint arXiv:2503.08311*, 2025.
- Sapkota, R., Roumeliotis, K. I., and Karkee, M. Ai agents vs. agentic ai: A conceptual taxonomy, applications and challenges. *arXiv preprint arXiv:2505.10468*, 2025.
- Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N., and Scialom, T. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023.
- Sentence-Transformers. static-retrieval-mrl-en-v1. <https://huggingface.co/sentence-transformers/static-retrieval-mrl-en-v1>. Hugging Face model card.
- Shavit, Y., Agarwal, S., Brundage, M., Adler, S., O'Keefe, C., Campbell, R., Lee, T., Mishkin, P., Eloundou, T., Hickey, A., et al. Practices for governing agentic ai systems. *Research Paper, OpenAI*, 2023.
- Sheng, Y., Zheng, L., Yuan, B., Li, Z., Ryabinin, M., Chen, B., Liang, P., Ré, C., Stoica, I., and Zhang, C. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pp. 31094–31116. PMLR, 2023.

- Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.
- Shridhar, M., Yuan, X., Côté, M.-A., Bisk, Y., Trischler, A., and Hausknecht, M. AlfworlD: Aligning text and embodied environments for interactive learning. *arXiv preprint arXiv:2010.03768*, 2020.
- Singh, J., Magazine, R., Pandya, Y., and Nambi, A. Agentic reasoning and tool integration for llms via reinforcement learning. *arXiv preprint arXiv:2505.01441*, 2025.
- SWE-agent. mini-swe-agent: The 100 line ai agent that solves github issues or helps you in your command line. <https://github.com/SWE-agent/mini-swe-agent>.
- Taulli, T. and Deshmukh, G. Haystack. In *Building Generative AI Agents: Using LangGraph, AutoGen, and CrewAI*, pp. 237–249. Springer, 2025.
- Telenti, A., Auli, M., Hie, B. L., Maher, C., Saria, S., and Ioannidis, J. P. Large language models for science and medicine. *European journal of clinical investigation*, 54(6):e14183, 2024.
- Thirunavukarasu, A. J., Ting, D. S. J., Elangovan, K., Gutierrez, L., Tan, T. F., and Ting, D. S. W. Large language models in medicine. *Nature medicine*, 29(8):1930–1940, 2023.
- Trivedi, H., Balasubramanian, N., Khot, T., and Sabharwal, A. Musique: Multihop questions via single-hop question composition. *Transactions of the Association for Computational Linguistics*, 10:539–554, 2022.
- Vellaisamy, P., Labonte, T., Chakraborty, S., Turner, M., Sury, S., and Shen, J. P. Characterizing and optimizing llm inference workloads on cpu-gpu coupled architectures, 2025. URL <https://arxiv.org/abs/2504.11750>.
- Vu, T., Iyyer, M., Wang, X., Constant, N., Wei, J., Wei, J., Tar, C., Sung, Y.-H., Zhou, D., Le, Q., et al. Freshllms: Refreshing large language models with search engine augmentation. *arXiv preprint arXiv:2310.03214*, 2023.
- Wang, B. and Komatsuzaki, A. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. <https://github.com/kingoflolz/mesh-transformer-jax>, May 2021.
- Wang, S., Xu, T., Li, H., Zhang, C., Liang, J., Tang, J., Yu, P. S., and Wen, Q. Large language models for education: A survey and outlook. *arXiv preprint arXiv:2403.18105*, 2024.
- Wang, W., Chen, Z., Chen, X., Wu, J., Zhu, X., Zeng, G., Luo, P., Lu, T., Zhou, J., Qiao, Y., et al. Visionllm: Large language model is also an open-ended decoder for vision-centric tasks. *Advances in Neural Information Processing Systems*, 36:61501–61513, 2023.
- Wolfram—Alpha. Wolfram—alpha instant calculators api: Reference & documentation. <https://products.wolframalpha.com/instant-calculators-api/documentation>.
- Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Liu, J., et al. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024.
- Xu, Y., Kong, X., Chen, T., and Zhuo, D. Conveyor: Efficient tool-aware llm serving with tool partial execution. *arXiv preprint arXiv:2406.00059*, 2024.
- Yang, H., Yue, S., and He, Y. Auto-gpt for online decision making: Benchmarks and additional opinions. *arXiv preprint arXiv:2306.02224*, 2023.
- Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.
- Yang, Z., Qi, P., Zhang, S., Bengio, Y., Cohen, W. W., Salakhutdinov, R., and Manning, C. D. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. *arXiv preprint arXiv:1809.09600*, 2018.
- Yao, S., Chen, H., Yang, J., and Narasimhan, K. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757, 2022.
- Yao, S., Zhao, J., Yu, D., Du, N., Shafraan, I., Narasimhan, K., and Cao, Y. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.
- Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, 2022.
- Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

Zhang, Y., Wei, C., Wu, S., He, Z., and Yu, W. Geogpt: Understanding and processing geospatial tasks through an autonomous gpt. *arXiv preprint arXiv:2307.07930*, 2023.

Zhao, W. X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2), 2023.

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E., et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36: 46595–46623, 2023.

Zhou, A., Yan, K., Shlapentokh-Rothman, M., Wang, H., and Wang, Y.-X. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*, 2023a.

Zhou, G., Hong, Y., and Wu, Q. Navgpt: Explicit reasoning in vision-and-language navigation with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 7641–7649, 2024.

Zhou, S., Xu, F. F., Zhu, H., Zhou, X., Lo, R., Sridhar, A., Cheng, X., Ou, T., Bisk, Y., Fried, D., et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023b.

Zhuo, T. Y., Vu, M. C., Chim, J., Hu, H., Yu, W., Widayarsi, R., Yusuf, I. N. B., Zhan, H., He, J., Paul, I., et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

A WORKLOAD IMPLEMENTATION DETAILS

A.1 Toolformer

We choose the same AI model (GPT-J 6B), calculation tool (WolframAlpha API (Wolfram—Alpha)) and mathematical benchmarks (ASDiv (Miao et al., 2021), SVAMP (Patel et al., 2021) and MAWPS (Koncel-Kedziorski et al., 2016)) for profiling as used in the original paper (Schick et al., 2023).

A.2 SWE-Agent

We choose mini-SWE-agent (SWE-agent), a research benchmarking version of SWE-agent using Qwen2.5-Coder-32B (Hui et al., 2024) model specifically suited for coding applications. We choose benchmarks derived from APPS (Hendrycks et al., 2021), BigCodeBench (Zhuo et al., 2024)

and DS-1000 (Lai et al., 2023), which are computationally intensive and can comprehensively showcase the CPU perspective.

A.3 Haystack

We choose ENNS top-5 retrieval using faiss FLAT (Douze et al., 2024) indexing and *static-retrieval-mrl-en-v1* embedding model (Sentence-Transformers) from C4 (Dodge et al., 2021) document corpus (305 GB english variant) for profiling using Natural Questions (NQ) (Kwiatkowski et al., 2019), HotpotQA (Yang et al., 2018) and TriviaQA (Joshi et al., 2017) benchmarks.

A.4 ChemCrow

We choose the same AI model (GPT-4-0613) and literature review tool based on Arxiv and Pubmed for profiling on chemistry QA benchmarks (nicotine, warfarin, caffeine, aspirin) inspired by the ChemCrow tasks. Since the model is proprietary, we use OpenAI API instead of local vLLM server used in all other workloads.

A.5 LangChain

We choose Google search API for web search, LexRank (Erkan & Radev, 2004) summarizer for summarization and GPT-OSS-20B model for LLM inference. We evaluate the workload on FreshQA (Vu et al., 2023), MusiQue (Trivedi et al., 2022) and QASC (Khot et al., 2020) benchmarks.

A.6 Software Environment

Our software environment includes PyTorch (version 2.8.0) and a local vLLM server (version 0.11.0) for LLM inference, except for Chemcrow benchmark which uses OpenAI API for GPT-4-0613 model inference. The workloads include langchain 0.3.27, haystack-ai 2.18.1, chemcrow 0.3.24, and mini-swe-agent 1.9.1.