MESSAGEFLOW: PROACTIVE MESSAGE-AWARE EXECUTION FOR EFFICIENT ON-DEVICE MULTI-AGENT SYSTEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

On-device multi-agent systems (MAS) powered by large language models (LLMs) face substantial response latency from chains of dependent LLM inference and tool calls. Agent-centric execution defers prompt processing until an agent is invoked, even when parts of its input are available earlier. We observe that evolving workflow messages can form stable, contiguous prompt prefixes that are ready for computation before control-flow dependencies are satisfied. Meanwhile, tool waits and autoregressive decoding often leave GPU resources underutilized, creating opportunities to overlap this early computation with ongoing execution. We present MessageFlow, a proactive message-aware execution framework that exploits these opportunities while preserving topology-defined agent invocation semantics. MessageFlow combines message update semantics, producer–consumer dependencies, and prompt ordering to identify reusable prefixes. Its runtime incrementally maintains and prioritizes the resulting prefill workloads, while an opportunity-aware GPU scheduler executes them during idle and decode-only periods and adapts to foreground demands to limit contention. The resulting key–value states are reused at invocation, moving part of prompt processing off the critical path. Implemented on LangGraph and SGLang, MessageFlow is evaluated on three representative workflows, four locally deployed LLMs, and three GPU platforms. Compared with state-of-the-art MAS execution systems, MessageFlow achieves up to 1.60× end-to-end speedup (1.32× on average), while increasing average GPU utilization from 57.8% to 89.6%.

1 INTRODUCTION

LLM-based multi-agent systems (MAS) are increasingly used to tackle complex tasks such as deep research (Zheng et al., 2025), financial analysis (Yu et al., 2024; Song et al., 2026), and complex question answering (Chen et al., 2025; Li et al., 2025a). Running these workflows locally on personal computers and workstations keeps sensitive documents, proprietary code, and private user data on-device while reducing reliance on cloud services (Song et al., 2024; Li et al., 2025b). However, achieving responsive local execution remains challenging, as MAS workflows often involve long chains of costly operations, including LLM inference and tool calls, whose latency accumulates along the execution path (Chen et al., 2025; Li et al., 2025a). One way to reduce this latency is to identify and exploit parallelism within workflow execution.

Agent-centric execution primarily exposes parallelism through the agent topology, which specifies the control-flow dependencies governing agent invocation (LangChain, n.d.; Qian et al., 2025; Hong et al., 2024). This permits agents on independent branches to run concurrently, but offers fewer opportunities along sequential paths. In the deep-research workflow in Figure 1, for example, a supervisor iteratively delegates research topics to researchers, which repeatedly invoke search tools and return their findings. The writer synthesizes the accumulated findings into a final report only after research completes, although some of its input is available much earlier (LangChain, 2025). Treating each agent invocation as the unit of scheduling defers prompt processing until the agent is invoked, even when part of that processing could be performed earlier.

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.

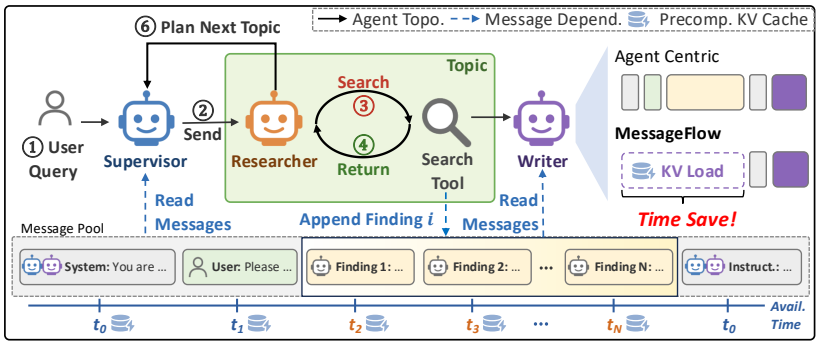

Figure 1: A deep-research workflow illustrating how early message availability enables proactive prefill of stable prompt prefixes before agent invocation.

Our key observation is that **readiness for agent invocation and readiness for partial prompt computation need not coincide.** Message dependencies identify the downstream consumers of each message, while each agent's prompt template specifies how these messages are assembled and ordered. Tracking message updates and stability together with this ordering allows us to identify a contiguous token prefix that will remain unchanged as the workflow progresses. Such a prefix can be proactively prefilled before the agent is formally invoked. For example, the writer's prompt template in Figure 1 specifies the following order: a system prompt, a user query, accumulated findings, and a final instruction. The system prompt is available from the start, and some prior work has exploited static system prompts for precomputation (Yu et al., 2026; Wadlom et al., 2026; Ye et al., 2025; Fang et al., 2026). The opportunity, however, extends to workflow messages that become available during execution: as findings are appended without changing earlier prompt tokens, they extend the stable prefix. Under causal attention, this prefix can be prefilled independently of later tokens, with its key–value (KV) states retained for reuse at invocation (Zheng et al., 2024). Although the final instruction is known in advance, it cannot be incorporated into the reusable prefix because its KV states depend on the preceding findings, which continue to grow.

Table 1: Average GPU utilization across three MAS workflows on NVIDIA RTX A6000.

Workflows	**Open Deep Research** (LangChain, 2025)	**PrimoAgent** (Botunac, 2025)	**LLMCompiler** (Kim et al., 2024)
GPU utilization (%)	60.42%	55.43%	57.45%

Such proactive computation is particularly attractive for local MAS execution because workflows naturally contain periods of GPU underutilization. As shown in Table 1, GPU utilization ranges from 55.43% to 60.42% across three MAS workflows on an NVIDIA RTX A6000 running Gemma3 27B. Waiting for external tool responses can leave the GPU idle (Kim et al., 2026), while autoregressive decoding can leave GPU compute resources underutilized (Agrawal et al., 2024; Kamath et al., 2025). These periods offer opportunities to overlap prefix precomputation with ongoing execution, potentially hiding part of the prefill cost and reducing end-to-end latency.

Building on these observations, we present MessageFlow, a proactive message-aware execution framework that turns this early input availability into computation overlap for efficient on-device MAS. MessageFlow keeps the agent topology authoritative for formal invocation while using message dependencies to prepare downstream prompt computation in advance. Realizing this opportunity poses three challenges. First, agent topology alone does not reveal which downstream prompt computations can safely begin before invocation. Proactive message awareness exposes the message evolution semantics and prompt dependencies needed to assess this readiness. Second, early computation becomes available incrementally across multiple agents, requiring continuous workload maintenance and prioritization. A message-aware MAS runtime tracks reusable prompt prefixes and organizes the resulting prefill work as the workflow evolves. Third, executing this prefill work can introduce resource contention that offsets the benefit of precomputation (Zhu et al., 2025). Opportunity-aware GPU scheduling exploits idle and decode-only periods while adapting proactive execution to changing foreground demands.

We implement MessageFlow atop LangGraph and SGLang and evaluate it across three representative real-world MAS workflows spanning deep research, financial analysis, and multi-hop question

answering. Our experiments cover four locally deployed LLMs—Gemma3 12B and 27B, Phi4 14B, and Qwen3.5 27B—on three GPU platforms with different compute and memory capabilities. Compared with state-of-the-art MAS execution systems, MessageFlow achieves up to 1.60× end-to-end speedup and 1.32× speedup on average, while increasing average GPU utilization from 57.8% to 89.6%.

2 PRELIMINARIES

2.1 AGENT TOPOLOGY AND MESSAGE DEPENDENCIES

An LLM-based multi-agent system (MAS) executes a workflow consisting of a set of agents $\mathcal{A}$. Its control-flow structure is described by an *agent topology*

$$G_A = (\mathcal{A}, E_A), \tag{1}$$

where an edge $(a_i, a_j) \in E_A$ represents a control-flow dependency from agent a_i to agent a_j. The agent topology therefore captures the execution dependencies that constrain when each agent may be formally invoked (LangChain, n.d.; Qian et al., 2025).

During workflow execution, agents communicate through a set of messages $\mathcal{M}$ whose values may evolve over time. We use S_t to denote the committed message state at time t, and $S_t[m]$ to denote the current value of message $m \in \mathcal{M}$ in this state. The producer–consumer relationships between agents and messages induce a bipartite *message dependency graph*

$$G_M = (\mathcal{A} \cup \mathcal{M}, E_P \cup E_C), \tag{2}$$

where $(a, m) \in E_P$ indicates that agent a may produce or update message m, and $(m, a) \in E_C$ indicates that message m is consumed by agent a. Unlike the agent topology, which captures control-flow dependencies, the message dependency graph captures how information is produced and propagated across agents.

The messages consumed by each agent constitute the dynamic inputs to its LLM invocation. When the agent is invoked, these messages are assembled according to its fixed prompt template to form the complete input context. We represent their ordering as

$$D_a = \langle m_{a,1}, m_{a,2}, \ldots, m_{a,K_a} \rangle, \tag{3}$$

where $m_{a,i} \in \mathcal{M}$ denotes the i-th consumed message according to its position in the agent's prompt template. We use $\mathrm{Render}_a(S_t)$ to denote the agent's native prompt construction procedure, which instantiates the template with the committed message values in S_t.

As workflow execution progresses, an increasingly long prefix of the prompt constructed for an agent may become stable before the agent is formally invoked. We denote by $P_a(t)$ the longest contiguous token prefix that is guaranteed to remain a prefix of any subsequent valid prompt constructed for agent a. As direct and exact reuse of prefix KV states under causal attention requires an unchanged token prefix (Zheng et al., 2024), $P_a(t)$ defines the portion of the agent's prompt that can be safely precomputed at time t.

2.2 PROACTIVE EXECUTION OBJECTIVE

Let π_0 denote conventional agent-centric execution. A proactive execution policy π prefills $P_a(t)$ before agent a is formally invoked and incrementally extends the retained inference state as the safe prefix grows. At invocation time, validated prefix states are reused, leaving only the remaining prompt tokens to be prefilled.

Our objective is to reduce end-to-end workflow latency $T_{\mathrm{E2E}}(\pi)$ by moving part of prompt processing ahead of agent invocation. Feasible policies operate on safe prefixes and respect device resource constraints. Formal agent invocations, routing, and message commits remain governed by the original workflow; proactive prefill prepares internal inference states without producing agent-visible outputs or updating workflow messages.

Achieving this objective requires answering two coupled questions: how to identify safe prompt prefixes as messages evolve, and when to execute their prefill so that otherwise underutilized compute resources can be exploited without offsetting the latency benefit through resource contention.

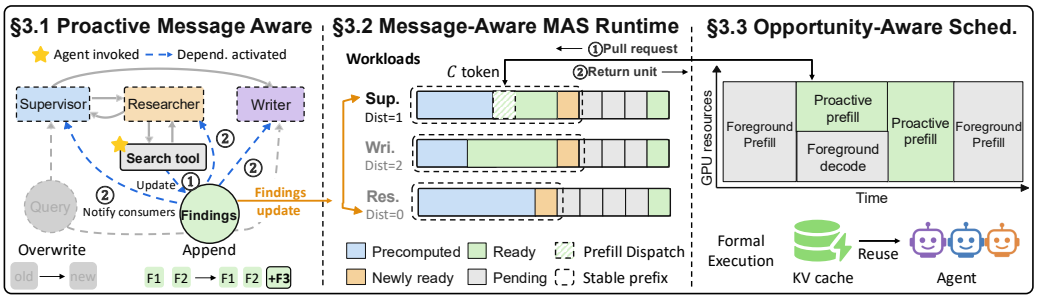

Figure 2: Overview of MessageFlow. Proactive message awareness (Sec 3.1) exposes message update semantics, producer–consumer dependencies, and prompt ordering, enabling the MAS runtime (Sec 3.2) to maintain stable prompt prefixes and prioritize incremental prefill workloads. The opportunity-aware GPU scheduler (Sec 3.3) requests prefill units from the runtime and executes them during idle or decode-only periods, adapting execution to foreground demands.

3 MessageFlow Framework

3.1 Proactive Message Awareness

Conventional MAS runtimes treat messages as passive, workflow-visible state that consuming agents read only when invoked. As illustrated in Figure 2, MessageFlow enables proactive message awareness through three explicit declarations: message categories describing how values may evolve, each agent's produced and consumed messages to construct G_M, and the order of consumed messages in each agent's prompt template to specify D_a. The agent topology G_A continues to govern formal agent invocation.

MessageFlow assigns each message m an update category,

$$u(m) \in \{\textsc{Append}, \textsc{Overwrite}, \textsc{Other}\},$$

to guide the assessment of whether its committed content is safe to reuse. An APPEND message preserves its existing content and only appends new content. Its committed content therefore remains valid under subsequent updates and can contribute to a consumer's safe prefix while the message continues to grow. An OVERWRITE message allows its current value to be replaced and becomes safe to reuse once no producer can modify it. OTHER covers update types for which MessageFlow cannot guarantee safe reuse; such messages are excluded from proactive computation. As additional message content becomes safe, MessageFlow uses G_M to identify the affected consumers and each consumer's D_a to locate that content within its prompt template. This information triggers the prefix reassessment described in Section 3.2.

3.2 Message-Aware MAS Runtime

The Message-Aware MAS Runtime turns evolving message availability into incremental prefill work for downstream agents. As illustrated in Figure 2, it tracks each candidate agent's safe prompt prefix and prefill progress, updating workloads and their priorities as messages arrive and the workflow advances. When GPU execution opportunities arise, it selects a suitable workload according to these priorities and dispatches a fixed-size prefill unit.

Maintaining Incremental Prefill Workloads. When a message update is committed, the runtime evaluates its safety using the message's update category and the current workflow progress. It then uses G_M to identify affected consumers and their prompt ordering D_a to reassess each consumer's longest safe prefix $P_a(t)$. For example, newly appended findings can extend the writer's safe prefix once preceding inputs are stable, whereas the final instruction following those findings remains excluded while the findings may still grow. Workflow progress also triggers reassessment when an `OVERWRITE` message becomes safe, even if its value has not changed.

Whenever $P_a(t)$ extends, the runtime creates or updates the corresponding workload while preserving previously computed KV states. Let $\ell_a(t)$ denote the number of tokens already prefilled for agent a. The workload therefore tracks the uncomputed suffix of the safe prefix, containing $|P_a(t)| - \ell_a(t)$

tokens. Further message arrivals can enlarge this suffix, while proactive execution advances $\ell_a(t)$. Maintaining both quantities allows the runtime to accumulate newly available work and execute it incrementally across successive GPU opportunities.

Prioritizing Candidate Workloads. Message updates may expose safe prefixes for multiple downstream agents, creating competing demands for limited proactive execution opportunities. To choose among these candidates, the runtime uses proximity to formal invocation as a priority heuristic. Let $\mathcal{F}_t$ denote the current execution frontier, consisting of agents scheduled for the next formal invocations. For each workload targeting agent a, the runtime computes

$$d_t(a) = \min_{v \in \mathcal{F}_t} \mathrm{dist}_{G_A}(v, a), \tag{4}$$

where $\mathrm{dist}_{G_A}(v, a)$ is the number of edges on the shortest path from v to a in the agent topology. Workloads are ordered by increasing $d_t(a)$, favoring agents closer to invocation in the topology. Conditional edges remain possible transitions; this heuristic does not predict future branch choices or loop counts. As workflow execution advances, the runtime updates the priorities and candidate set. Distance-zero targets leave the proactive queue because their remaining computation is handled by formal invocation. Targets that have completed execution and will not be invoked again are also removed.

Fine-Grained Prefill Dispatch. MessageFlow executes only one proactive workload at a time to limit resource contention. The priority ordering determines which workload to select when the GPU scheduler requests work. However, executing its entire remaining suffix would delay reconsideration of newly available or higher-priority candidates. The runtime therefore divides proactive prefill into execution units of C tokens, allowing workload selection to be revisited after each unit.

For each request, the runtime scans candidates in priority order and selects the first workload satisfying

$$|P_a(t)| - \ell_a(t) \geq C. \tag{5}$$

It returns the next C uncomputed tokens of that workload's safe prefix. Candidates below this threshold remain queued for further prefix growth; if none qualifies, no work is dispatched. After the unit completes, the runtime retains the resulting KV cache and advances $\ell_a(t)$. At the next execution opportunity, it selects again using the updated prefixes, progress, and priorities. When an agent is formally invoked, the retained prefix computation is reused and any remaining prompt tokens are processed by the formal request. Once a target has completed execution and will not be invoked again, its request handle and retained KV cache are released.

3.3 Opportunity-Aware GPU Scheduling

The MAS runtime identifies and prioritizes safe prefill workloads, but executing them immediately may cause resource contention that offsets the benefit of precomputation. MessageFlow therefore uses the GPU scheduler to pull work from the runtime during GPU execution bubbles or periods of resource underutilization (Figure 2). The scheduler allocates GPU resources according to the foreground workload and adjusts or suspends proactive execution as foreground demands change (Figure 3).

Execution Opportunity Detection. As shown in Figure 3, the scheduler monitors both active and queued foreground work to identify opportunities for proactive execution. It distinguishes three execution states: IDLE, DECODEONLY, and BUSY. IDLE indicates that no foreground prefill or decode work is executing or waiting in the queue, allowing proactive work to use the available GPU resources. DECODEONLY indicates that foreground inference consists only of decoding, providing an opportunity to overlap proactive prefill with foreground execution. Any executing or queued foreground prefill places the scheduler in BUSY, during which no new proactive work is admitted.

When an eligible opportunity appears, the scheduler first resumes suspended proactive work, if available. Otherwise, it requests work from the MAS runtime, which returns an eligible execution unit according to the priority and token-budget rules in Section 3.2. If no workload qualifies, the scheduler repeats the request at a fixed polling interval while the opportunity remains open, rechecking the execution state before each attempt.

Table 2: Specifications of the evaluation hardware.

Device	GPU Arch.	SMs	Memory	GPU Clock
A6000	Ampere	84	48 GB GDDR6	1.80 GHz
Spark	Blackwell	48	128 GB LPDDR5x	3.00 GHz
A4500	Ampere	56	20 GB GDDR6	1.65 GHz

Table 3: SM allocation for proactive prefill.

Device	G3 12B	Phi4 14B	G3 27B	Q3.5 27B
A6000	52/84	36/84	60/84	12/84
Spark	24/48	28/48	36/48	16/48
A4500	26/56	16/56	OOM	OOM

Prefill–Decode Resource Partitioning. The two opportunity types require different resource allocations. During IDLE, proactive prefill can use all available GPU compute resources. During DECODEONLY, it shares the device with foreground decoding, so its execution must be coordinated to limit interference with foreground progress.

Mixed prefill–decode batching combines prefill and decoding within the same execution iteration, coupling decode progress to the execution time of the mixed batch. A large prefill unit can therefore increase the interval between successive decode steps. To decouple their execution, MessageFlow partitions the GPU's streaming multiprocessors (SMs) between proactive prefill and foreground decoding (Lin et al., 2026; Chen et al., 2026; Gao et al., 2026). The two workloads execute concurrently on disjoint SM sets, allowing successive decode steps to proceed without waiting for the entire proactive unit to finish. Let $R_P(t)$ and $R_D(t)$ denote the SM sets allocated to proactive prefill and foreground decoding, respectively, and let R_{GPU} denote all allocatable SMs. The allocations satisfy

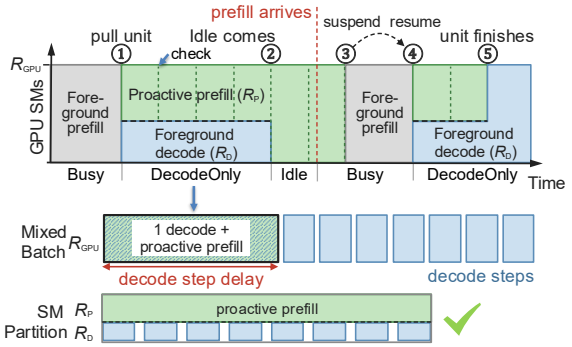

Figure 3: Opportunity-aware GPU scheduling.

$$R_P(t) \cap R_D(t) = \varnothing, \qquad R_P(t) \cup R_D(t) \subseteq R_{\mathrm{GPU}}. \tag{6}$$

We profile each model–device pair to select an appropriate SM partition ratio.

Adapting to Foreground Changes. The foreground execution state may change before a proactive unit completes, altering the resources available to it. MessageFlow therefore checks the execution state after every fixed number of model layers within each unit. These checkpoints allow the scheduler to respond to foreground changes without waiting for the entire C-token unit to finish.

If a foreground prefill request is waiting, the scheduler suspends proactive computation at the next checkpoint and allows the foreground request to proceed. The suspended work retains its execution state and resumes when a suitable opportunity becomes available. The scheduler also adjusts SM allocations at these checkpoints according to active and queued foreground work. A transition from IDLE to DECODEONLY activates SM partitioning, whereas a transition from DECODEONLY to IDLE assigns all available SMs to proactive prefill. When proactive work finishes and only foreground decoding remains, decoding receives all available SMs.

These layer-level checkpoints allow adaptation within an execution unit. Once the unit completes, its KV state and progress are retained. If a suitable execution opportunity remains, the scheduler requests the next unit from the MAS runtime, which selects work using the latest workload state and priorities.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

We evaluate MessageFlow on three workloads: *Open Deep Research* (LangChain, 2025) (hierarchical web research) with 20 English Deep Research Bench questions (Du et al., 2026); *PrimoAgent* (Botunac, 2025) (sequential trading analysis) with four stocks over ten trading days; and *LLMCompiler* (Kim et al., 2024) (planner–tool–joiner QA) with 50 randomly sampled HotpotQA bridge questions (Yang et al., 2018). Baselines include *LangGraph* (LangChain, n.d.), an agent-centric MAS framework; *Static*, which precomputes and reuses KV states for fixed system prompt, following a common optimization in prior MAS systems (Wadlom et al., 2026; Ye et al., 2025); and

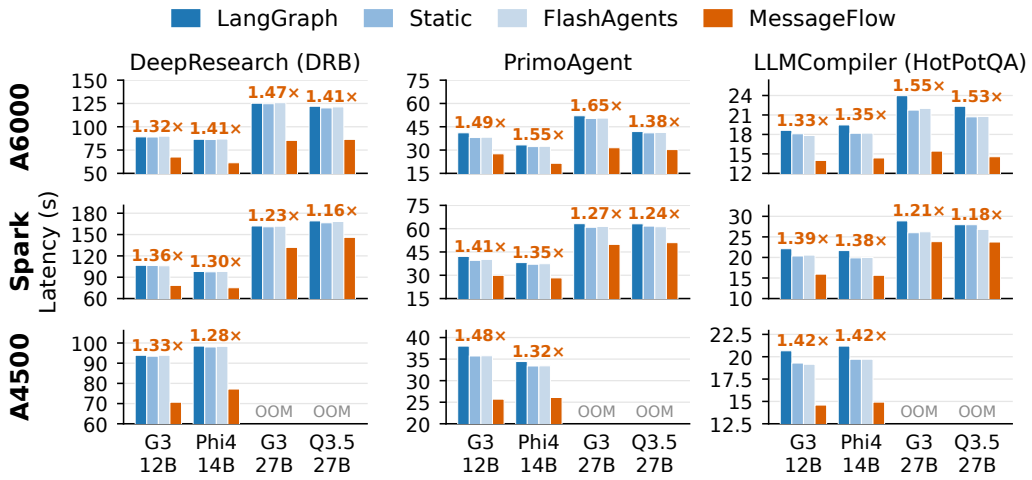

Figure 4: Mean latency across workflows, models, and devices. Annotations show MessageFlow's speedup over LangGraph; OOM denotes insufficient device memory.

Table 4: Prefill precomputation coverage on A6000 with Gemma3-27B.

	LangGraph	Static	FlashAgents	**MessageFlow**
Deep Research	0%	4.80%	4.95%	**77.29%**
PrimoAgent	0%	6.86%	7.31%	**94.15%**
LLMCompiler	0%	19.86%	20.45%	**81.94%**

FlashAgents (Fang et al., 2026), which combines static caching with incremental downstream prefill from streamed upstream outputs. We locally deploy INT4-quantized Gemma3 12B/27B (Gemma Team, 2025), Phi4 14B (Abdin et al., 2024), and Qwen3.5 27B (Qwen Team, 2026) on NVIDIA RTX A6000, DGX Spark, and RTX A4500 (Table 2), with A4500 limited to Gemma3 12B and Phi4 14B by GPU memory. Table 3 lists proactive prefill SM allocations for DECODEONLY. We measure response-start latency from input submission to the first token of the final MAS response (Fang et al., 2026; Wang et al., 2026); unless otherwise stated, speedups in Section 4 are relative to LangGraph. Appendix B provides additional experimental details.

4.2 OVERALL PERFORMANCE

Figure 4 shows that MessageFlow achieves the lowest latency across all 30 evaluated configurations, providing 1.16–1.65× speedups over LangGraph (1.37× arithmetic mean). Average speedups are 1.33×, 1.41×, and 1.38× for Deep Research, PrimoAgent, and LLMCompiler, respectively. The largest gain occurs on PrimoAgent with Gemma3 27B on A6000, reducing latency from 52.07 s to 31.48 s (39.5%). MessageFlow also consistently outperforms Static and FlashAgents, particularly on Deep Research, where both baselines offer little improvement over LangGraph. With Gemma3 27B on A6000, MessageFlow reduces latency to 85.38 s, compared with 124.70 s for Static and 125.84 s for FlashAgents. This advantage is consistent with exploiting stable prompt content that becomes available as workflow messages accumulate.

Table 4 reports the fraction of prefill work completed proactively before agent invocation. On A6000 with Gemma3 27B, MessageFlow achieves 77.29–94.15% coverage across the three workflows, compared with 4.80–19.86% for Static and 4.95–20.45% for FlashAgents. Tracking stable content as messages accumulate exposes more prefill work for overlap, helping explain MessageFlow's latency advantage.

4.3 ABLATION STUDY

Table 5 evaluates MessageFlow's scheduling components across three workflows on A6000 with Gemma3 27B. The full design achieves the lowest latency throughout. Replacing opportunity-aware scheduling with eager dispatch, which submits proactive work as soon as it becomes available, increases latency by 16.0–49.6%. On Deep Research, latency rises from 85.38 s to 127.77 s, exceeding LangGraph's 125.25 s. Thus, poorly timed proactive execution can offset its benefits.

Table 5: Ablation results (latency in seconds).

Variant	DeepRes.	Primo.	LLMCom.
LangGraph	125.25	52.07	23.97
MessageFlow	**85.38**	**31.48**	**15.43**
- Fine-grained dispatch	90.89	36.44	16.26
- Opportunity-aware	127.77	43.35	17.90
- Idle utilization	108.61	41.77	19.49
- Resource partition	119.59	41.25	20.27
- Foreground adaptation	108.45	38.91	16.61

Disabling idle-period proactive prefill while retaining overlap with decoding increases latency by 26.3–32.7%, confirming the additional benefit of idle windows. Replacing resource partitioning with mixed prefill–decode batching increases latency by 31.0–40.1%, supporting partitioned execution to reduce interference with foreground decoding. Replacing fine-grained dispatch with a single submission of all ready prefill work increases latency by 5.4–15.8%, most notably on PrimoAgent. Disabling foreground adaptation, including proactive-work preemption for foreground requests and dynamic resource reallocation, increases latency by 7.6–27.0%, with the largest penalty on Deep Research. These results support incremental dispatch and foreground adaptation, with benefits varying across workflows.

4.4 GPU Utilization Analysis

Figure 5 compares GPU utilization on PrimoAgent with Gemma3 27B on A6000. LangGraph exhibits frequent idle gaps and periods of partial utilization. MessageFlow sustains utilization near 100% for longer intervals, increasing average GPU utilization from 55.4% to 90.3%, a gain of 34.9 percentage points. This aligns with MessageFlow's scheduling strategy: proactive prefill occupies GPU idle periods and overlaps with foreground decoding when spare compute capacity is available. These results support MessageFlow's ability to exploit execution bubbles for downstream prefill, helping translate early message availability into lower workflow latency.

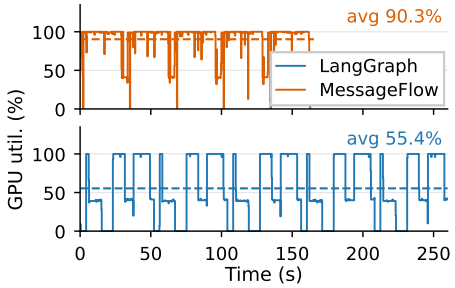

Figure 5: GPU utilization comparison of LangGraph and MessageFlow.

Table 6: Perplexity under different prefill schemes.

Prefill scheme	G3 12B	Phi4 14B	G3 27B	Q3.5 27B
Unchunked	9.2476	8.3850	7.6649	6.7728
Chunk 2048	9.2489	8.3851	7.6648	6.7726
Chunk 4096	9.2488	8.3848	7.6649	6.7748
MessageFlow	**9.2489**	**8.3853**	**7.6656**	**6.7732**

Table 7: LLMComp. HotpotQA F1.

Model	LangGraph	**MessageFlow**
Gemma3 12B	69.44%	69.17% (**-0.27 pp**)
Phi4 14B	58.96%	59.76% (**+0.80 pp**)
Gemma3 27B	75.25%	75.29% (**+0.04 pp**)
Qwen3.5 27B	75.23%	75.62% (**+0.39 pp**)

4.5 Numerical Error Analysis

MessageFlow preserves the underlying computation in exact arithmetic. However, floating-point addition is not associative, so changes in accumulation order may introduce small rounding differences (PyTorch, n.d.). Prefill partitioning may change this order, as can conventional chunked prefill. We evaluate WikiText-2 (Merity et al., 2017) perplexity (PPL) on 8,192-token inputs, each randomly split into five contiguous segments for MessageFlow's proactive prefill. Table 6 compares MessageFlow with unchunked prefill and chunked prefill with 2,048- and 4,096-token chunks. Across all four models, MessageFlow's PPL differs from unchunked prefill by at most 1.3×10^{-3}, with relative differences below 0.015%. Conventional chunked prefill also shows small deviations, confirming that numerical variation is not unique to proactive execution. On 500 HotpotQA examples, LLMCompiler with MessageFlow yields answer F1 changes of -0.27 to $+0.80$ percentage points relative to LangGraph (Table 7). Together, these results indicate limited impact on aggregate predictive performance and downstream task quality in the evaluated settings, without implying identical generated outputs.

4.6 Sensitivity to Network Latency

The workflows rely on web search, whose latency varies with network conditions and service load. To assess MessageFlow's robustness to variations in tool-response latency, we vary the delay from

Table 8: Sensitivity to Firecrawl response latency on PrimoAgent (LangGraph / MessageFlow).

Firecrawl latency (s)	Gemma3 12B		Phi4 14B		Gemma3 27B		Qwen3.5 27B	
	Latency(s)	Speedup	Latency(s)	Speedup	Latency(s)	Speedup	Latency(s)	Speedup
0.5	35.76/24.77	1.44	27.97/21.17	1.32	46.34/32.71	1.42	36.52/31.43	1.16
1.0	38.20/25.96	**1.47**	30.50/21.31	1.43	49.07/32.83	1.49	39.25/31.62	1.24
1.5	40.51/28.59	1.42	32.95/22.15	**1.49**	51.87/32.07	**1.62**	41.48/31.53	1.32
2.0	42.63/31.00	1.38	35.47/24.60	1.44	54.01/34.34	1.57	44.18/31.80	1.39
2.5	45.35/33.31	1.36	37.93/27.10	1.40	56.56/36.35	1.56	46.50/32.16	**1.45**

0.5 s to 2.5 s on PrimoAgent across four models running on an NVIDIA RTX A6000. Table 8 shows that MessageFlow achieves 1.16–1.62× speedups over LangGraph. Longer tool waits generally provide more opportunities for proactive prefill. For Qwen3.5 27B, LangGraph latency increases from 36.52 s to 46.50 s, while MessageFlow remains nearly unchanged (31.43–32.16 s), raising the speedup from 1.16× to 1.45×. Gemma3 12B and Phi4 14B instead peak at intermediate delays, as proactive work is finite and additional waiting cannot be further hidden. MessageFlow consistently reduces latency across all tested models and delays.

4.7 Portability to Ascend 910B3

To assess portability beyond NVIDIA GPUs, we evaluate MessageFlow on Huawei Ascend 910B3, a non-NVIDIA accelerator. Since SM partitioning is unavailable on this platform, we use a simplified implementation that performs proactive prefill only during tool waits. On LLMCompiler with Qwen3.5 27B, MessageFlow reduces latency from 18.51 s with LangGraph to 14.92 s, yielding a 19.39% reduction and a 1.24× speedup.

5 Related Work

MAS Runtime Execution. General-purpose MAS frameworks LangGraph (LangChain, n.d.) and AutoGen (Wu et al., 2024) expose concurrency through agent invocation dependencies. Topology-aware serving system TOPAS (Ni et al., 2026) coordinates cross-workflow request scheduling without exploiting early message availability for prompt computation. FlashAgents (Fang et al., 2026) overlaps downstream prefill with streamed upstream outputs through topology-defined interactions, without explicitly targeting tool-induced GPU idle periods. MessageFlow explicitly abstracts message dependencies and evolution, enabling proactive computation beyond inter-agent streaming while preserving topology-defined invocation semantics.

MAS KV Cache Reuse. KVFlow (Pan et al., 2025) and TOPAS (Ni et al., 2026) optimize cache management to reduce eviction-induced misses and recomputation when concurrent workflows compete for limited capacity. Our on-device setting emphasizes early preparation of reusable KV states over cross-user cache contention. Radix caching (Zheng et al., 2024) reuses computed prefixes; static-prompt precomputation covers fixed inputs (Wadlom et al., 2026; Ye et al., 2025). Neither alone exploits early availability of evolving workflow messages; MessageFlow identifies and precomputes their stable prefixes, expanding reusable computation at downstream invocation.

6 Conclusion

We presented MessageFlow, a proactive message-aware execution framework for on-device multi-agent systems. By tracking message evolution and prompt dependencies, MessageFlow identifies stable prompt prefixes and incrementally prefills them during GPU idle and decode-only periods, while preserving topology-defined agent invocation semantics. Across three workflows, four LLMs, and three GPU platforms, MessageFlow achieves up to 1.60× end-to-end speedup (1.32× on average) over state-of-the-art MAS execution systems and increases average GPU utilization from 57.8% to 89.6%. These results demonstrate that early message availability exposes computation opportunities beyond agent-level parallelism, enabling more efficient local MAS execution.

AI USE STATEMENT

In this work, we used generative AI tools for translation, language editing, code debugging, literature searches, and manuscript consistency checks. We did not use generative AI tools to implement the proposed method, develop theoretical results or proofs, design experiments, or interpret experimental results. Data generation and processing tasks requiring disclosure were not applicable to this work. The authors reviewed AI-assisted text to ensure that it preserved the intended technical meaning, verified the cited sources, and reviewed and tested code changes suggested during debugging. We take responsibility for the final content of this work, including all text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J. Hewett, Mojan Javaheripi, Piero Kauffmann, James R. Lee, Yin Tat Lee, Yuanzhi Li, Weishung Liu, Caio C. T. Mendes, Anh Nguyen, Eric Price, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Xin Wang, Rachel Ward, Yue Wu, Dingli Yu, Cyril Zhang, and Yi Zhang. Phi-4 technical report. Technical Report MSR-TR-2024-57, Microsoft Research, 2024. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2024/12/P4TechReport.pdf.

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 117–134, 2024.

Joshua Bakita and James H. Anderson. Hardware compute partitioning on NVIDIA GPUs. In *Proceedings of the 29th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 54–66, 2023. doi: 10.1109/RTAS58335.2023.00012.

Ive Botunac. Implementation of a multi-agent artificial intelligence system for financial trading decision-making process. *Oeconomica Jadertina*, 15(2):90–115, 2025. doi: 10.15291/oec.4863.

Yukang Chen, Weihao Cui, Han Zhao, Ziyi Xu, Xiaoze Fan, Xusheng Chen, Yangjie Zhou, Shixuan Sun, Bingsheng He, and Quan Chen. Towards high-goodput LLM serving with prefill-decode multiplexing. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2026. doi: 10.1145/3779212.3790236.

Zehui Chen, Kuikun Liu, Qiuchen Wang, Jiangning Liu, Wenwei Zhang, Kai Chen, and Feng Zhao. MindSearch: Mimicking human minds elicits deep AI searcher. In *The Thirteenth International Conference on Learning Representations*, 2025.

Mingxuan Du, Benfeng Xu, Chiwei Zhu, Licheng Zhang, Xiaorui Wang, and Zhendong Mao. DeepResearch Bench: A comprehensive benchmark for deep research agents. In *The Fourteenth International Conference on Learning Representations*, 2026.

Taosong Fang, Zhen Zheng, Zhengzhao Ma, Yaojie Lu, Hongyu Lin, Xianpei Han, and Le Sun. FlashAgents: Accelerating multi-agent LLM systems via streaming prefill overlap. In *Proceedings of Machine Learning and Systems*, 2026.

Lei Gao, Chaoyi Jiang, Hossein Entezari Zarch, Daniel Wong, Mark D. Hill, and Murali Annavaram. DuetServe: Harmonizing prefill and decode for LLM serving via adaptive GPU multiplexing. In *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, 2026.

Gemma Team. Gemma 3 technical report. Technical report, Google DeepMind, 2025. URL https://storage.googleapis.com/deepmind-media/gemma/Gemma3Report.pdf.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven 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.

Aditya K Kamath, Ramya Prabhu, Jayashree Mohan, Simon Peter, Ramachandran Ramjee, and Ashish Panwar. POD-Attention: Unlocking full prefill-decode overlap for faster LLM inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 897–912, 2025. doi: 10.1145/3676641.3715996.

Jiin Kim, Byeongjun Shin, Jinha Chung, and Minsoo Rhu. The cost of dynamic reasoning: Demystifying AI agents and test-time scaling from an AI infrastructure perspective. In *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2026. doi: 10.1109/HPCA68181.2026.11408569.

Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. An LLM compiler for parallel function calling. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 24370–24391, 2024.

LangChain. Open deep research, 2025. URL https://www.langchain.com/blog/open-deep-research.

LangChain. LangGraph runtime, n.d. URL https://docs.langchain.com/oss/python/langgraph/pregel. Accessed September 25, 2026.

Ao Li, Yuexiang Xie, Songze Li, Fugee Tsung, Bolin Ding, and Yaliang Li. Agent-oriented planning in multi-agent systems. In *The Thirteenth International Conference on Learning Representations*, 2025a.

Yilong Li, Jingyu Liu, Hao Zhang, Badri Narayanan Murali Krishnan, Utkarsh Sharma, Shuai Zhang, Yijing Zeng, Jayaram Raghuram, and Suman Banerjee. PalmBench: A comprehensive benchmark of compressed large language models on mobile platforms. In *The Thirteenth International Conference on Learning Representations*, 2025b.

Zejia Lin, Hongxin Xu, Guanyi Chen, Zhiguang Chen, Yutong Lu, and Xianwei Zhang. Bullet: Boosting GPU utilization for LLM serving via dynamic spatial-temporal orchestration. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2026. doi: 10.1145/3779212.3790135.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *The Fifth International Conference on Learning Representations*, 2017.

Hongqiu Ni, Han Tian, Chi Zhang, Guopeng Li, and Haisheng Tan. TOPAS: Workflow-aware prefix-state scheduling for multi-agent LLM serving. *arXiv preprint arXiv:2608.25523*, 2026.

Zaifeng Pan, Ajjkumar Dahyalal Patel, Yipeng Shen, Zhengding Hu, Yue Guan, Wan-Lu Li, Lianhui Qin, Yida Wang, and Yufei Ding. KVFlow: Efficient prefix caching for accelerating LLM-based multi-agent workflows. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-4208.

PyTorch. Numerical accuracy, n.d. URL https://docs.pytorch.org/docs/stable/notes/numerical_accuracy.html. Accessed September 25, 2026.

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 *The Thirteenth International Conference on Learning Representations*, 2025.

Qwen Team. Qwen3.5: Towards native multimodal agents, 2026. URL https://qwen.ai/blog?id=qwen3.5.

Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. PowerInfer: Fast large language model serving with a consumer-grade GPU. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pp. 590–606, 2024. doi: 10.1145/3694715.3695964.

Zifan Song, Kaitao Song, Guosheng Hu, Ding Qi, Junyao Gao, Xiaohua Wang, Dongsheng Li, and Cairong Zhao. Trade in minutes! rationality-driven agentic system for quantitative financial trading. In *The Fourteenth International Conference on Learning Representations*, 2026.

Noppanat Wadlom, Junyi Shen, and Yao Lu. Efficient LLM serving for agentic workflows: A data systems perspective. *Proceedings of the ACM on Management of Data*, 4(3), 2026.

Junlin Wang, Jue Wang, Zhen Xu, Ben Athiwaratkun, Bhuwan Dhingra, Ce Zhang, and James Zou. Staircase streaming for low-latency multi-agent inference. In *ICML 2026 Workshop on Resource-Adaptive Foundation Model Inference (AdaptFM)*, 2026.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed 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.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2369–2380, 2018. doi: 10.18653/v1/D18-1259.

Hancheng Ye, Zhengqi Gao, Mingyuan Ma, Qinsi Wang, Yuzhe Fu, Ming-Yu Chung, Yueqian Lin, Zhijian Liu, Jianyi Zhang, Danyang Zhuo, and Yiran Chen. KVCOMM: Online cross-context KV-cache communication for efficient LLM-based multi-agent systems. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-0605.

Shan Yu, Junyi Shu, Yuanjiang Ni, Kun Qian, Xue Li, Yang Wang, Jinyuan Zhang, Ziyi Xu, Shuo Yang, Lingjun Zhu, Ennan Zhai, Qingda Lu, Jiarong Xing, Youyou Lu, Xin Jin, Xuanzhe Liu, and Harry Xu. Pythia: Exploiting workflow predictability for efficient agent-native LLM serving. *arXiv preprint arXiv:2604.25899*, 2026.

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 *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-4354.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-2000.

Yuxiang Zheng, Dayuan Fu, Xiangkun Hu, Xiaojie Cai, Lyumanshan Ye, Pengrui Lu, and Pengfei Liu. DeepResearcher: Scaling deep research via reinforcement learning in real-world environments. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 414–431, 2025. doi: 10.18653/v1/2025.emnlp-main.22.

Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Tian Tang, Qinyu Xu, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Ziren Wang, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. NanoFlow: Towards optimal large language model serving throughput. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pp. 749–765, 2025.

A Appendix Outline

This appendix supplements the main paper with additional experimental details, correctness analysis, and implementation examples. It is organized as follows:

- Appendix B provides additional details on the experimental setup.
- Appendix C discusses prefix safety and the scope of the correctness guarantees.
- Appendix D describes the implementation of MessageFlow.
- Appendix E explains the selection of SM partitions.
- Appendix F illustrates workflow integration through an LLMCompiler case study.
- Appendix G provides a theoretical comparison with FlashAgents.

B Detailed Experimental Setup

This section supplements Section 4.1 with the workload selection, comparison conditions, and measurement protocol.

Workloads and datasets. Our evaluation covers hierarchical research, sequential financial analysis, and planned multi-hop question answering, exposing different patterns of agent interaction and message accumulation.

Open Deep Research (LangChain, 2025) produces research reports through a supervisor–researcher hierarchy: the supervisor delegates topics, researchers iteratively invoke search tools, and a writer synthesizes the accumulated findings. We use 20 English tasks from Deep Research Bench (Du et al., 2026), whose expert-authored research questions span 22 domains and require gathering and synthesizing information from multiple sources. We exclude Chinese tasks because limited Chinese-language support in some evaluated models can lead to execution-trace divergences that complicate trace validation. This workload combines nested reasoning–tool loops with progressively growing report context.

PrimoAgent (Botunac, 2025) generates a SELL, HOLD, or BUY recommendation with an associated confidence score through successive data collection, technical analysis, news analysis, and portfolio-management stages. Later agents consume market evidence and analyses produced upstream, forming a sequential decision pipeline. Our evaluation inputs cover Apple, NVIDIA, Microsoft, and Tesla over ten trading days, with historical market data and company-related news providing the evidence for each stock–date analysis.

LLMCompiler (Kim et al., 2024) answers questions through a planner–tool–joiner pipeline: the planner decomposes a question into retrieval tasks, tools gather evidence, and the joiner synthesizes an answer. Across compared modes, we retain its original streaming planner execution: tool calls are incrementally parsed from the planner's output and dispatched once their arguments and dependencies are available, without waiting for planner decoding to finish. We randomly sample 50 bridge questions from HotpotQA (Yang et al., 2018) for latency evaluation. HotpotQA provides Wikipedia-based multi-hop questions and annotated supporting facts; bridge questions connect evidence across articles through an intermediate entity. These inputs exercise dependencies between retrieval steps and the accumulation of evidence for answer synthesis. A separate evaluation uses 500 HotpotQA examples to measure answer F1.

Baselines. *LangGraph* submits each agent's LLM request at its topology-defined invocation, without proactive prefill. *Static* prepares the fixed instruction prefix before online execution and retains its KV states for subsequent requests. Precomputing and reusing KV states for fixed prompt prefixes is a common optimization in MAS, used in systems such as KVComm (Ye et al., 2025), Helium (Wadlom et al., 2026), and Pythia (Yu et al., 2026). This prefix ends at the first input-dependent position: a constant instruction appearing after dynamic content cannot be included. Static preparation is excluded from online latency. The *FlashAgents* comparison augments static preparation with downstream prefill during upstream decoding. MessageFlow additionally identifies stable prefixes from committed workflow messages and executes their prefill during idle and

decode-only opportunities. We use SGLang for LLM inference and implement the SM partitioning with `libsmctrl` (Bakita & Anderson, 2023), which applies TPC masks to CUDA streams to assign disjoint compute resources to proactive prefill and foreground decoding. Table 3 reports the model–device-specific allocations; their selection is discussed in Appendix E.

SGLang configuration. We keep radix caching enabled across all compared methods. We use greedy decoding (`temperature=0`) to eliminate sampling randomness and reduce variation in generated outputs and subsequent workflow execution. All other SGLang parameters use their default values and remain identical across compared methods.

Latency and speedup. Response-start latency spans input submission to the first token of the final workflow response. It includes intervening agent computation, tool waits, and runtime overhead, while excluding the remaining streamed final answer. For each evaluation item and method, we perform one warm-up run followed by three measured runs and average the three measurements to obtain the item's latency. We flush the inference cache between runs, including after warm-up, to prevent cached states from carrying over across repetitions. We compare execution traces across methods and exclude the rare samples with divergent agent or tool execution from all compared methods' latency results. The reported latency is the mean of these per-item averages over the common set of trace-consistent evaluation inputs. For each workflow–model–device configuration, speedup is the ratio of LangGraph's mean latency to the compared method's mean latency on the same input set. The overall mean speedup is the arithmetic mean of these per-configuration ratios over the 30 evaluated configurations.

Prefill precompute coverage. Coverage measures the fraction of formal input tokens proactively prefilled before the target invocation begins. For formal invocation i, let N_i be its complete input length and P_i the longest matching prefix whose proactive prefill has completed beforehand. We aggregate coverage as $100 \sum_i P_i / \sum_i N_i$, counting each invocation separately and each prefix token once across incremental stages. Queued or unfinished work is excluded, and invocation-time waits for in-flight stages remain part of online latency. Coverage measures completed precomputation; actual KV reuse additionally requires the corresponding states to remain available at invocation. Ordinary radix-cache hits reuse prefixes computed for earlier requests and do not count as proactive prefill for the target invocation. Thus, LangGraph has zero precompute coverage because it initiates no proactive prefill, even with radix caching and prefix reuse.

Low coverage may yield no net speedup if the overhead of maintaining and dispatching proactive work offsets the saved prefill time. Coverage measures input tokens rather than time saved: latency benefits also depend on prefill cost, execution overlap, and runtime overhead. It should therefore be interpreted alongside end-to-end latency, rather than as a direct predictor of speedup.

C Safe Prefix Identification and Correctness

MessageFlow separates message-level eligibility from token-level validation: declared dependencies identify candidate prefixes, while exact token checks govern incremental continuation and reuse. Safety is assessed against subsequent valid prompts constructed for the agent. For agents that appear repeatedly in a loop, prefix eligibility is reassessed as workflow execution progresses.

Message stability. The runtime tracks committed message values and the execution frontier using the declared producer–consumer relations in G_M. For APPEND messages, it checks that each committed serialization preserves the previous value as a prefix. For OVERWRITE messages, it conservatively checks whether a producer other than the target is reachable from the frontier and can subsequently reach the target in G_A; such a producer prevents early use of the current value. Conditional edges and loops remain possible paths, without predicting routing outcomes. Frontier changes trigger reassessment even without a value change. Missing messages and OTHER updates are ineligible. Nested parallel appends are buffered in logical dispatch order; rollback or a divergent outer commit invalidates affected precomputation.

Contiguous prompt construction. The planner scans consumed messages in prompt order D_a, stopping at the first missing or ineligible field. Workflow-specific prompt builders then determine

the reusable boundary within the rendered input. An eligible growing message may extend this boundary, but text following its open append region must remain excluded until that region closes. For example, a writer can prefill committed findings while withholding the final instruction that follows them. Prefix and formal requests share the model's tokenizer and chat template. The adapters use complete chat-message boundaries or omit trailing boundary tokens when rendering an open append region. These are conservative construction rules, not a general proof that arbitrary string appends preserve tokenization; exact token validation remains necessary.

Validated continuation and reuse. Let Q_a be the token sequence whose staged prefill has completed for a target invocation, and let $X_a = \mathrm{Tok}(\mathrm{Render}_a(S_{t_a}))$ be its complete input at invocation time t_a, where Tok denotes the serving tokenizer. Each incremental stage must extend the previously retained token sequence exactly. At formal invocation, the runtime removes queued work for that target, waits for its in-flight stage, and reuses request-owned inference states, including KV states and any convolution history and recurrent states required by hybrid models, only if

$$Q_a = X_a[1{:}k], \qquad k = |Q_a|. \tag{7}$$

On a match, it submits only the remaining suffix through the retained request handle. If staged prefill fails or the token prefix differs, the client cancels the staged request and submits the complete input. Session-based reuse instead submits the complete token sequence to the serving engine for cache matching. A matching prefix hash or serialized message alone never authorizes continuation from retained inference states.

Correctness argument. For causal attention, token-prefix equality makes staged prefill equivalent to processing the same prefix within the full prompt in exact arithmetic, provided model weights, positional encodings, attention masks, and initial inference state are identical. At each layer, a prefix token depends only on preceding tokens and itself; later prompt tokens cannot alter its hidden representation or KV states. Induction over layers and successive prefix extensions therefore establishes equality of the retained prefix states. Processing the remaining suffix then yields the same full-prompt state and next-token logits as ordinary prefill.

The argument extends to causal hybrid models such as Qwen3.5 when the complete inference state is retained. Let $z_i = F(z_{i-1}, u_i)$ denote a layer's state update for token input u_i, where z_i includes its convolution history and recurrent state. Under the same model and positional settings, identical initial states and token inputs in the same order yield identical subsequent states by induction, provided segmented execution applies the same updates and preserves the complete state at each boundary, without resetting or replaying updates. Combining this token-wise induction with the layer-wise argument above establishes equality of the prefix inference states and subsequent logits in exact arithmetic for both attention-only and hybrid models.

The full-input fallback also preserves the formal request. Proactive work produces no agent-visible output and does not commit workflow messages or select branches. With fixed external responses, deterministic decoding and routing, and the same logical commit order, induction over formal invocations consequently preserves the workflow-visible trace.

Scope of the guarantee. Early eligibility relies on accurate message declarations and compatible prompt builders; the generic field scan does not establish safety for arbitrary reducers or renderers. Failed validation can waste computation without authorizing incorrect reuse. The argument above also does not imply bitwise-identical floating-point execution: different prefill partitions may perturb logits and generated outputs, motivating the numerical and task-quality evaluations in Section 4.

D Implementation Details

We implement MessageFlow using LangGraph 0.5.4 and SGLang 0.5.16, with a serving integration based on Bullet. LangGraph remains responsible for agent invocation, tool execution, and workflow state, while the serving backend manages proactive prefill and foreground generation. Separate prefill and decode workers run on the same GPU and share model-weight storage. The main implementation challenge is preserving a consistent computation as messages arrive incrementally, GPU execution is interrupted, and a future agent becomes ready to run. The following sections describe these transitions; prefix eligibility and correctness are covered in Appendix C.

D.1 Committed Message Integration

Observing consistent workflow state. The runtime observes message updates at LangGraph's superstep boundaries, where the updates from concurrently executing nodes have been combined into a committed state. A node returning a result does not by itself establish the value that a downstream agent will consume: another node may contribute to the same field before the superstep completes. MessageFlow therefore reads the committed state together with the next execution frontier, namely the nodes that can execute next. It updates message values and versions from this snapshot before reconsidering future prefill candidates. Keeping the state and frontier aligned prevents planning from partially merged messages or outdated execution information.

Nested workflows require an additional mechanism because waiting for an outer node to finish can hide useful intermediate results. We expose committed appends from nested executions through an observation interface, while preserving the order in which those results will enter the outer state. For example, if two research units are dispatched in order, results from the second remain buffered until the first closes, even when the second finishes earlier. The visible prefix can thus grow during nested execution without depending on task completion order. If a nested execution fails, or the final outer commit differs from the observed sequence, the runtime invalidates the affected precomputation and restores its view from the authoritative workflow state. These observations serve only the optimization and do not independently invoke downstream agents.

D.2 Resumable Proactive Prefill

Unless otherwise stated, we use $C = 4096$ tokens per proactive prefill unit and check execution state every four model layers.

Keeping a prefill request alive across stages. A proactive prefix may grow across message arrivals, so completing a prefill stage must not end the inference request. A staged request handle retains the cache allocation and completed prefix across calls. After a non-final stage, the server acknowledges completion without generating text and parks the request, preserving its slot and KV states. Later stages submit only new tokens through the same handle; the server restores the token-to-cache mapping and extends the allocation. This preserves incremental progress without recomputing earlier stages or relying solely on a subsequent cache lookup.

At formal invocation, the client removes queued proactive work for the target and waits for any in-flight stage, preventing concurrent prefix extension during foreground continuation. It then performs the exact token-prefix check in Appendix C and submits the remaining suffix through the retained handle. The backend transitions from the retained prefix state to next-token prediction during formal continuation. If prefill failed or the prefix no longer matches, the client cancels the staged request and submits the complete prompt. Cleanup also cancels unconsumed handles to release resources.

Suspending and restoring unfinished computation. Pausing within a prefill stage requires retaining the current layer position, intermediate hidden states, and model-specific residual tensors and inputs. For example, Gemma3 retains the position embeddings used by its global and local attention layers. The backend executes layers in groups and resumes from the saved layer without replaying completed layers. Before resumption, it rebuilds attention metadata for the request's sequence and cache layout, which intervening foreground work may have replaced.

Layer checkpoints cooperate with the serving engine's token-chunk scheduler. Launched GPU kernels finish, and stream dependencies ensure subsequent operations observe their writes. When foreground prefill arrives, proactive execution pauses at the next layer checkpoint, allowing the foreground request to proceed even within an unfinished token chunk. The backend retains the partial chunk's execution state and cache mapping, resuming from the saved layer when an opportunity becomes available. Chunk bookkeeping advances only after all layer groups for that chunk complete.

D.3 Coordination Between Prefill and Decode Workers

Tracking asynchronous execution opportunities. The prefill worker detects opportunities from queued work and GPU execution state. CPU submission alone is insufficient, since CUDA calls can return before their kernels finish. The backend records completion events on the execution stream

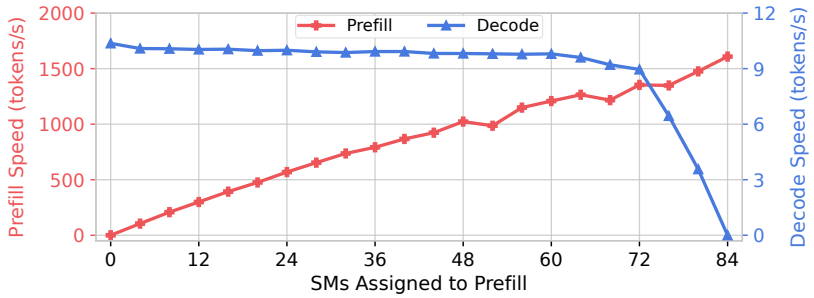

Figure 6: Prefill and decode speeds under different SM allocations for Gemma3 27B on NVIDIA RTX A6000.

and clears activity state only after they complete. It also marks decoding as active before dispatching a request to the decode worker, covering the interval until the first decode step. These mechanisms prevent pending GPU work or in-transit requests from being mistaken for idle intervals.

During idle or decode-only opportunities, a background client repeatedly requests eligible work from the MAS runtime. An empty response leaves the opportunity open, since new messages may soon make a prefix ready. Each pull processes a bounded prefill unit, after which remaining work is reconsidered alongside new candidates. Foreground prefill arrivals close the opportunity and receive priority at safe scheduling boundaries. SM allocations follow the active phases and model–device-specific settings in Appendix E.

Transferring cache state and ownership. Prefill-to-decode handoff must preserve both computed state and storage ownership. The workers share cache pools but use separate CUDA contexts, so producer-stream ordering alone does not synchronize the consumer. Before handoff, the prefill worker synchronizes outstanding GPU writes, sends request information and cache-slot mappings to the decode worker, and removes the request from its local batch without freeing shared slots. Decode continues from the populated cache and releases resources through normal completion. This prevents premature cache reuse and concurrent continuation of the same request by both workers.

For Qwen3.5, handoff also includes Gated DeltaNet convolution history and recurrent state, which are required in addition to attention KV to represent the completed prefix. The backend keeps these states resident and transfers their request mappings and sequence-tracking information alongside the attention cache. During layer-wise prefill, each visited layer updates its state normally; resumption uses the saved layer position without copying a separate checkpoint at every boundary. Allocation includes recurrent-state slots and rolls back partial reservations if other request resources cannot be allocated. This extends the same request-lifetime discipline to hybrid models with reusable inference state beyond KV.

E Hyperparameter Selection for SM Partitioning

Prefill and decode respond differently to SM allocation. Prefill processes multiple tokens in parallel, enabling weight reuse and high arithmetic intensity, and generally benefits from more SMs. At small batch sizes, autoregressive decoding is typically memory-bandwidth bound and can sustain a similar generation rate with fewer SMs. However, allocating too few SMs to decoding limits its progress, requiring a balance between prefill throughput and foreground decode speed.

We select the partition through offline profiling for each model–device pair, allowing at most a 10% decode speed loss. Let D_0 denote the decode-only rate with all SMs, and $D(s)$ the decode rate when proactive prefill uses s SMs concurrently with decoding on the remaining SMs. We seek high prefill throughput among allocations satisfying $D(s) \geq 0.9D_0$.

Figure 6 illustrates this trade-off for Gemma3 27B on NVIDIA RTX A6000. Prefill throughput generally increases with more SMs, while decode speed remains relatively stable up to approximately 60 prefill SMs and drops sharply beyond approximately 72. We allocate 60 of the A6000's 84 SMs

LangGraph

```
builder.add_node(PLANNER, ...)
builder.add_node(EXECUTE, ...)
builder.add_node(JOINER, ...)
```

Agent Topology

```
builder.add_edge(START, PLANNER)
builder.add_edge(PLANNER, EXECUTE)
builder.add_conditional_edges(...)
builder.add_edge(JOINER, END)
builder.compile()
```

MessageFlow

```
messages: Annotated[
    list[CompilerMessage], APPEND,
    accumulate=True,]
builder.add_msgflow_node(PLANNER, ...
    consumes=("messages",),
    produces=("messages",))

builder.add_msgflow_node(EXECUTE, ...
    consumes=("messages",),
    produces=("messages",))

builder.add_msgflow_node(JOINER, ...
    consumes=("messages",),
    produces=("messages",),
   prompt_builder=...)
```

Figure 7: LLMCompiler implementation in LangGraph and MessageFlow. MessageFlow retains the agent topology while adding message category, message dependencies, and prompt template.

to proactive prefill and 24 to decoding, achieving approximately 1,200 prefill tokens/s with less than 10% decode speed loss.

The selected partition for DECODEONLY remains fixed for each model–device pair throughout the corresponding experiments; allocations in other execution states follow Section 3.3. Table 3 reports 60/84 prefill SMs for Gemma3 27B but only 12/84 for Qwen3.5 27B on A6000, motivating model–device-specific profiling rather than a universal partition ratio.

F LLMCompiler Workflow and MessageFlow Implementation

We use LLMCompiler to illustrate MessageFlow integration. In our HotpotQA configuration, the Planner streams a dependency graph of search and bridge-entity extraction tasks. The executor dispatches available tasks once their dependencies are satisfied and appends observations to the shared message history. After all required tasks finish, the Joiner synthesizes the evidence into a final answer. The prompts and trace below illustrate a multi-hop example.

Figure 7 compares the LangGraph and MessageFlow implementations. MessageFlow preserves the agent topology while declaring message inputs and outputs through `consumes` and `produces`, the shared `messages` field as `APPEND`, and the Joiner's prompt construction through `prompt_builder`. These declarations expose stable prefixes as observations arrive, enabling proactive prefill during execution opportunities before formal invocation. The Joiner still waits for all required evidence before generating an answer.

System Prompts

Planner. Given a user query, construct a complete static DAG plan for the user's HotPotQA bridge Question.
The plan may contain only three actions:
`search(entity)`: search Wikipedia for an entity and return the results.
`extract_bridge_entity(observation, hint, expected_type)`: deterministically extract a named bridge entity from a preceding observation. This operation uses ordinary program logic and does not invoke an LLM.
`join()`: collect completed observations and invoke the final Joiner once after all planned dependencies finish.
Dependencies must reference preceding task IDs (e.g., `$1`).
Terminate with `<END_OF_PLAN>`.
Followed by three few-shot demonstrations: `[examples omitted]`

Joiner. Answer the user question using the provided Wikipedia observations.
First provide a **Thought** that reasons over the observations in 1–2 sentences. Then provide exactly one action:

```
Finish(answer)
```

The answer must be **short and contain a single item**. Terminate with `<END_OF_RESPONSE>`.

Planner Execution

User Query. Who founded the Oxford college where the author of The Hobbit studied as an undergraduate?

Planner Output.

```
1. search("The Hobbit author")
2. extract_bridge_entity($1, hint="author of The Hobbit",
     expected_type="person")
3. search("$2 undergraduate college at Oxford")
4. extract_bridge_entity($3, hint="Oxford undergraduate college",
     expected_type="organization")
5. search("$4 founder")
6. join()
   <END_OF_PLAN>
```

Tool Execution Trace

Tool 1: Search

```
search("The Hobbit author")
→  The Hobbit, also known as There and Back Again, was written by J.
R. R. Tolkien and first published in 1937.  ...
```

Tool 2: Bridge-Entity Extraction

```
extract_bridge_entity($1, hint="author of The Hobbit",
  expected_type="person")
→ J. R. R. Tolkien
```

Tool 3: Search

```
search("J. R. R. Tolkien undergraduate college at Oxford")
→    Tolkien began his studies at Exeter College, Oxford, in October
1911.  His original subject was classics, ...
```

Tool 4: Bridge-Entity Extraction

```
extract_bridge_entity($3,
  hint="Oxford undergraduate college",
  expected_type="organization")
→ Exeter College, Oxford
```

Tool 5: Search

```
search("Exeter College, Oxford founder")
→ Founded by Walter de Stapeldon, a Devon man who rose from a humble
background to become Bishop of Exeter and Treasurer of England under
Edward.  ...
```

Joiner Output

Thought: The author of *The Hobbit* is J. R. R. Tolkien, who studied at Exeter College, Oxford. The college was founded by Walter de Stapeldon.

Action: `Finish(Walter de Stapeldon)`

`<END_OF_RESPONSE>`

G Theoretical Comparison with FlashAgents

FlashAgents (Fang et al., 2026) accelerates multi-agent workflows by streaming upstream-generated tokens to overlap downstream prefill with upstream decoding. This section examines potential reasons for its limited effectiveness in our target workflows, focusing on missed tool-derived context and the dependence of prefill work on upstream token generation.

Missed tool context. FlashAgents' generation-driven prefill mechanism leaves an important source of downstream computation unaddressed: text returned by external tools. Web search can produce large volumes of retrieved content, which can dominate downstream prompt length and prefill cost

in search-intensive workflows. These observations arrive as tool results rather than tokens streamed from an upstream LLM, so their availability is not explicitly captured by FlashAgents' streaming triggers. Consequently, the mechanism can miss opportunities to precompute a substantial portion of the downstream context, even when stable search results are already available and subsequent tool calls leave the GPU idle. Overlapping prefill of generated tokens alone does not address this remaining workload. MessageFlow explicitly tracks committed tool observations, identifies the portions that extend stable prompt prefixes, and schedules their prefill during subsequent tool waits or decode-only periods.

Decode-limited prefill. FlashAgents couples downstream precomputation to upstream token generation: generated tokens accumulate in a buffer and trigger incremental prefill once a threshold is reached or generation finishes. This design imposes a producer-rate bottleneck. When downstream prefill is much faster than upstream decoding, each accumulated chunk is processed in a short burst, followed by a longer wait for new tokens. The streaming mechanism therefore cannot continuously supply prefill work to exploit otherwise available GPU compute capacity. Larger chunks prolong the accumulation delay, while smaller chunks increase execution frequency and may reduce computational efficiency. Adjusting the chunk size cannot eliminate the underlying work-supply bottleneck.

This mismatch also limits latency reduction. For a single stream with generation time T_d and prefill time T_p, assuming fixed computation costs, negligible overhead, and no resource contention, overlap alone provides at most

$$S_{\text{overlap}} \leq \frac{T_d + T_p}{\max(T_d, T_p)}.$$

When $T_p \ll T_d$, this bound approaches $1 + T_p/T_d$, so even complete hiding of prefill produces limited acceleration. The weakness is therefore structural: generation-driven precomputation restricts both the supply of new work and the latency that can be hidden. MessageFlow addresses this limitation by exposing additional prefill work from stable workflow messages, including tool observations, rather than relying solely on tokens arriving from an active generation stream.