MULTI-AGENT LLM SERVING IS A FULL-STACK PROBLEM: A SMALL STEP TOWARDS ALGORITHM–SYSTEM CO-DESIGN

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models are increasingly deployed as multi-agent systems (MAS) that coordinate many LLM calls through branching, debate, and iterative refinement. MAS algorithms evolve quickly and are tuned for accuracy, whereas serving systems evolve slowly and cannot be specialized to each new workflow, so MAS run on general-purpose engines that are blind to workflow structure. We take a step toward algorithm–system co-design. We cast MAS algorithms as typed workflow graphs $G = (N, E, Op)$ with a phase-aware cost model, turning opaque multi-agent executions into analyzable serving workloads. On this representation we derive two families of optimization: *static patterns*, recurring motifs that a runtime accelerates without changing what the workflow computes, and *dynamic strategies*, graph rewrites that trade bounded accuracy for efficiency. Measurements on H100 GPUs show where each pays off. Shared-prefix reuse cuts time-to-first-token by up to $2.42\times$, yet end-to-end latency stays at parity because decode takes over 95% of it, as the cost model's Amdahl bound of at most $1.03\times$ predicts. Rewrites that remove decode work do pay off. Consensus-based early exit runs 3.1–$5.1\times$ faster within 0.02 accuracy on four math benchmarks, and sparse expert dispatch cuts LLM calls $6\times$ for 2.8 points of pass@1. On code generation, a single direct call already matches a five-expert ensemble. Workflow structure tells a runtime what it *can* optimize, and the phase profile tells it what is *worth* optimizing.

1 INTRODUCTION

Large language models (LLMs) are increasingly deployed not as single-shot predictors but as *multi-agent systems* (MAS): structured programs that decompose a task, assign specialized roles, and coordinate many interdependent LLM calls through branching, voting, debate, and iterative refinement. Role-based pipelines such as MetaGPT (Hong et al., 2024), conversational frameworks such as AutoGen (Wu et al., 2024), tool orchestrators such as HuggingGPT (Shen et al., 2023), and graph-centric frameworks such as GPTSwarm (Zhuge et al., 2024) and LangGraph (LangChain, 2024) have made this paradigm central to practice, and recent work even searches over workflow structure automatically (Zhang et al., 2025b;a). Multi-agent workflows now underpin applications ranging from software engineering to deep research.

This capability comes at a serving cost that current infrastructure is ill-equipped to absorb. We trace the root cause to a gap between two research communities that rarely meet. The first half concerns pace and specialization. New coordination structures such as debate, reflection, tree search, role pipelines, and difficulty-aware routing appear far faster than serving hardware and systems can be specialized to them. No single stack can be tailored to every MAS variant, so these workloads run on general-purpose engines whose lack of specialization shows up as low serving efficiency.

The second half of the gap concerns objectives. Algorithm research optimizes almost exclusively for task quality and treats the serving system as a black box, so workflows are full of choices that are innocuous for accuracy yet expensive for hardware: running every agent on the same frontier model, re-encoding the full conversation on every turn, or running a fixed number of debate rounds regardless of difficulty. Never tuned for efficiency, these workflows leave large headroom. As we

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.

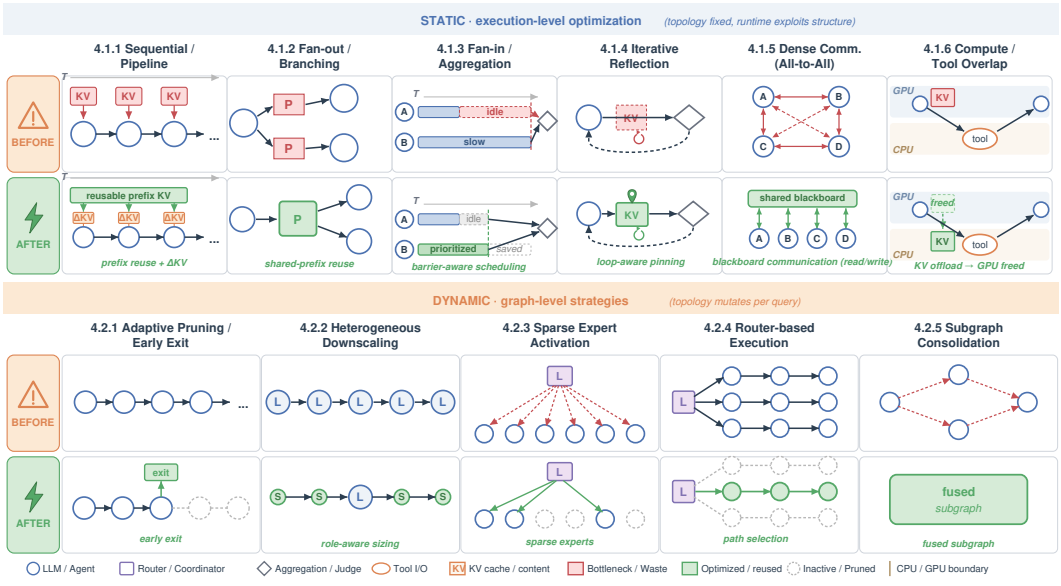

Figure 1: The two families of optimization we develop, each as a before/after pair. **Static patterns** (top) leave the workflow's computation untouched and let the runtime exploit its structure through scheduling and memory management. **Dynamic strategies** (bottom) rewrite the graph to cut computation at a bounded accuracy cost.

show, small structural changes remove up to 67% of a workflow's LLM calls or cut them 6× at a bounded accuracy loss.

This paper gives algorithm designers a system-level view of their workflows, a step toward *algorithm–system co-design* for LLM serving. The central obstacle is diversity: absent a common representation, each MAS must be analyzed from scratch. We therefore cast a broad class of MAS algorithms as a typed graph $G = (N, E, Op)$ over computation nodes, tool nodes, data-dependency edges, and control operators, with a phase-aware cost model that prices every node by its prefill and decode work. On this representation we pursue the two directions of Figure 1. *Static patterns* are recurring motifs, such as chains, fan-out, fan-in, and cycles, that a runtime accelerates *without changing what the workflow computes*. *Dynamic strategies* rewrite the graph, e.g., by early exit or model downscaling, to cut computation at a bounded accuracy cost. In summary, we make three contributions.

- **A unified workflow graph** $G = (N, E, Op)$ under which existing and emerging MAS algorithms reduce to a small set of canonical structures, with a phase-aware cost model whose Amdahl bound predicts the end-to-end payoff of an optimization (Section 3).
- **A catalogue of six static patterns and five dynamic strategies**, each tied to the graph property that induces it, the cost it reduces, and the condition under which it applies, which covers two structurally opposite real systems, Multi-Agent Debate and MetaGPT (Section 4).
- **Measurements on H100 GPUs** across five math benchmarks and MBPP. Prefix reuse cuts time-to-first-token by up to 2.42× but leaves end-to-end latency at parity, as the cost model predicts. Rewrites that remove decode work do pay off, with early exit up to 5.1× faster than the full debate at measured accuracy costs, and a single direct call matches a five-expert ensemble on code (Section 5).

2 Related Work

LLM and agentic serving systems. Serving systems improve scheduling and memory management through paged KV allocation in vLLM (Kwon et al., 2023), radix-tree prefix reuse in SGLang (Zheng et al., 2024), chunked prefills in Sarathi-Serve (Agrawal et al., 2024), prefill–decode disaggregation (Zhong et al., 2024; Patel et al., 2024), cold-start mitigation in ServerlessLLM (Fu et al., 2024),

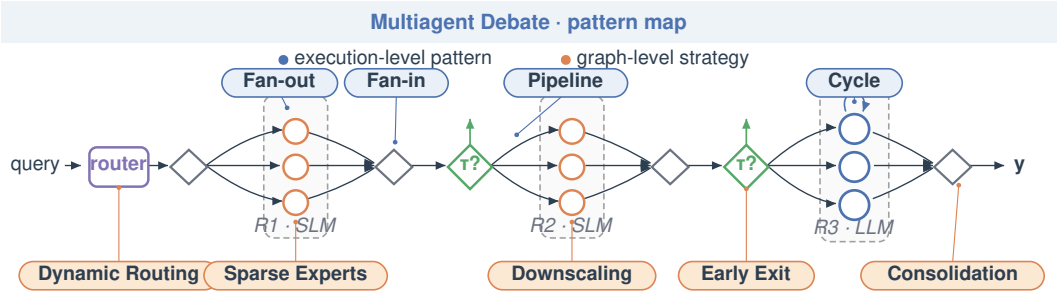

Figure 2: Multi-Agent Debate as a pattern map. Each round alternates execution-level motifs (fan-out, fan-in, pipeline, self-cycle) with graph-level strategies, escalating from small models in early rounds to a large model in the last.

and programmable KV control in Pie (Gim et al., 2025). Agent-aware systems go further. Parrot schedules DAGs of semantic variables (Lin et al., 2024), InferCept manages a request's KV while an external tool runs (Abhyankar et al., 2024), KVFlow evicts and prefetches KV along an agent step graph (Pan et al., 2025), Agentix schedules whole agent programs rather than individual calls (Luo et al., 2026a), KVCOMM reuses KV across agents whose shared context sits at different offsets (Ye et al., 2025a), and further work optimizes agentic pipelines at the batch and plan level (Shen et al., 2025). Closest in spirit, Ayo parallelizes and pipelines the primitive-level dataflow graph of an LLM application (Tan et al., 2025). Each system exploits one structural property inside one engine and preserves what the workflow computes, which supports our thesis that workflow structure must be surfaced to the backend. We ask the complementary questions of which properties a workflow exposes, which quality-affecting rewrites it admits, and which of them are worth exploiting for a given phase profile.

Agentic workflows and their automation. GPTSwarm formalizes agents as DAGs and optimizes their connectivity with reinforcement learning (Zhuge et al., 2024). AFlow searches over workflows of reusable operators with MCTS (Zhang et al., 2025b), MaAS samples query-dependent subnetworks from an agentic supernet (Zhang et al., 2025a), and DAAO adapts depth, operators, and model assignment to query difficulty (Su et al., 2026). These methods mutate subgraphs but optimize accuracy or monetary cost, not backend objectives. A complementary line asks when coordination pays off at all. A science-of-scaling study derives predictive principles over canonical topologies (Kim et al., 2025), OneFlow simulates many homogeneous workflows with a single agent and KV reuse (Xu et al., 2026), and AgentArk distills multi-agent debate into one model's weights (Luo et al., 2026b). Our cost model adds the serving side of these consolidation decisions.

3 Formalization: Unified Workflow Graph

Intuitively, a multi-agent workflow is a flowchart: boxes are LLM or tool calls, arrows carry data between them, and control primitives such as "vote" or "branch" decide what runs next. We make this precise as a typed directed *workflow graph*

$$G = (N, E, Op), \tag{1}$$

with nodes N, data-dependency edges E, and control operators Op. Making these first-class turns an opaque multi-step execution into an analyzable system workload, and the graph's *structure* and its *rewrites* are exactly what the two optimization families of Section 4 exploit. Figure 2 draws a real system, Multi-Agent Debate, in this notation.

Nodes N come in three kinds. A *computation node* $n = (M, P, C)$ is one model call (model M, prompt P, decoding config C) and is where GPU cost is paid. A *tool node* $n = T$ is an external, non-LLM call (web search, code execution, file retrieval). A *router node* $n = (R, P_r, C_r)$ uses a lightweight function R (a small model, a classifier, or a rule) to decide which downstream branches fire. The conditional operator below enacts that decision.

Edges $E \subseteq N \times N \times \mathcal{D}$ carry both connectivity and the data payload $\mathcal{D}$ (message text, retrieved passages, scores), that is, what B receives from A. We tag each edge with a coarse transfer type that ties structure to serving cost: C2G (CPU→GPU, e.g., a prompt becoming prefill input), G2C (GPU→CPU, e.g., outputs returning to a controller), or in-place. These tags already expose KV duplication, GPU residency during tool stalls, and barriers.

Operators Op regulate control flow without adding compute nodes. Two recur throughout. The *ensemble* operator $Op_{\text{ENSEMBLE}} : (d_1, \ldots, d_K) \mapsto d$ merges K upstream results (voting, debate synthesis) and imposes a fan-in barrier that waits on the slowest branch. The *conditional* operator $Op_{\text{CONDITION}} : (\mathcal{S}, d) \mapsto \{\text{next nodes}\}$ branches, loops, stops early, or routes from the workflow state $\mathcal{S}$, and underlies the dynamic strategies of Section 4.2.

Execution semantics. A node fires once all of its incoming edges carry data, so independent nodes may run concurrently. Cycles such as reflection loops and debate rounds are templates that the runtime unrolls, and conditional operators decide at run time which unrolled nodes execute. Every execution is therefore a DAG, and an analysis of G covers all of them. The types make serving properties checkable. For example, two computation nodes can share KV cache only if they run the same model M and their prompts begin with an identical token sequence.

A phase-aware cost model. GPU cost is paid at computation nodes in two phases. A node n on model M_n first prefills its p_n uncached prompt tokens in one compute-bound pass, then decodes its d_n output tokens one memory-bound step at a time, so its latency is

$$t_n = T^{\text{pre}}_{M_n}(p_n) + T^{\text{dec}}_{M_n}(d_n). \tag{2}$$

Workflow latency is the longest path through the executed DAG weighted by t_n and by tool latencies, GPU time is $\sum_n t_n$, and KV residency integrates each node's cache size over its time on the GPU. Because decode advances one token per step, long generations let T^{dec} dominate. If an optimization accelerates a phase that takes a fraction f of latency by a factor s and leaves the rest unchanged, Amdahl's law limits the end-to-end speedup to

$$S_{\text{e2e}} = \frac{1}{(1-f) + f/s} \;<\; \frac{1}{1-f}. \tag{3}$$

This bound organizes our results. An optimization pays off end to end only in proportion to the share of time spent in the phase it targets, so the same pattern can be decisive for one workload and irrelevant for another (Section 5.2).

4 Pattern-Driven System Optimizations

The two families of optimization follow directly from the graph $G = (N, E, Op)$ and are separated by their semantic impact. A *static pattern* is a recurring subgraph $G' \subseteq G$ whose *structure* (a chain, a high-degree node, a barrier, a cycle) already determines a serving inefficiency and its remedy. The remedy changes when, where, and from which cache the work runs, but never a prompt, a model, or an edge, so every node computes the same function as before. A *dynamic strategy* instead *rewrites* the graph, reassigning a node's model in N, pruning or adding edges in E, or inserting a control operator in Op. It changes what is computed, so its quality cost must be measured. In terms of Equation 2, static patterns shrink the uncached prompt p_n, the idle time on the critical path, or the KV residency, whereas dynamic strategies remove whole nodes or make each token cheaper and so cut decode as well as prefill.

4.1 Static Patterns: Execution-Level Optimization

Figure 1 (top) summarizes six recurring execution patterns. Each maps a structural invariant of G to a cost reduction that requires no change to the algorithm, provided the stated condition holds.

Pattern I: Sequential / pipeline execution. A chain $N_0 \to \cdots \to N_T$ of computation nodes passes each agent's output to the next, as in planner–executor and retrieval-generation pipelines. When turn t re-encodes its whole history of L_t tokens, naive serving prefills $\sum_t L_t$ prompt tokens. Prefix KV

reuse (SGLang (Zheng et al., 2024)) prefills only the Δ_t new tokens of each turn, $\sum_t \Delta_t$ in total, and leaves the outputs unchanged. The saving requires consecutive calls on the same model whose prompts extend a token-identical prefix. A growing conversation satisfies this, whereas roles with different system prompts do not unless the prompt layout places the shared history first.

Pattern II: Fan-out / branching. A node with out-degree K dispatches a shared context to K parallel agents, as in debate, Tree-of-Thought expansion, and ensemble reasoning (Du et al., 2024; Yao et al., 2023a). Over a shared prefix of length L, naive serving prefills KL tokens and launches K prefills at once. Shared-prefix reuse via RadixAttention (SGLang (Zheng et al., 2024)) prefills the prefix once, for $L + K\delta$ tokens with δ the per-agent suffix, and Sarathi-Serve (Agrawal et al., 2024) interleaves chunked prefills with decoding to absorb the burst. The reuse opportunity grows with the branching factor K.

Pattern III: Fan-in / aggregation. K parallel branches converge on an Op_{ENSEMBLE} node (voting, debate synthesis, verification). This is a barrier: the merge starts at $\max_i t_i$, so tail latency is set by the slowest branch while finished branches wait with their KV still resident. Because the dependency is explicit in G, a DAG-aware scheduler (Parrot (Lin et al., 2024)) can prioritize the requests on the critical path and release the memory of finished branches early, without changing the topology.

Pattern IV: Iterative reflection. A self-cycle on one node critiques and revises its own output across rounds (Shinn et al., 2023). The system prompt and prior context form a prefix that is *invariant* across iterations, yet is re-encoded every round. Loop-aware KV persistence (Pie (Gim et al., 2025)) pins the invariant prefix, so each round prefills only its new tokens.

Pattern V: Dense communication (all-to-all). K agents exchange messages over an all-to-all edge set, repeatedly consuming one another's outputs. Copying every message into every recipient's prompt moves $O(K^2)$ messages per round. A shared blackboard (Parrot's semantic variables (Lin et al., 2024)) stores each message once and passes $O(K)$ references instead. This removes copy and transfer overhead, but each agent must still prefill every token it reads, so the LLM-side cost falls only with selective reads, summarization, or KV reuse across readers (KVCOMM (Ye et al., 2025a)).

Pattern VI: Compute / tool overlap. Computation nodes alternate with tool nodes across G2C/C2G edges (ReAct (Yao et al., 2023b), browser, and retrieval agents). While a tool runs, the calling node's KV sits idle on the GPU for the tool's unbounded latency. The runtime can keep, discard, or swap that KV to host memory (InferCept (Abhyankar et al., 2024), building on vLLM's paged swapping (Kwon et al., 2023)). Swapping pays off when the stall outlasts the round trip of the KV over the host link, and recomputation when the context is short, so the right choice depends on the tool-latency profile that G makes visible.

4.2 Dynamic Strategies: Graph-Level Optimization

Dynamic strategies *rewrite* G to reduce the number of invocations, the communication volume, or the active compute, each with an explicit cost–quality trade-off (Figure 1, bottom, and Appendix B).

Strategy I: Adaptive subgraph pruning (early exit). Insert a conditional operator Op_{EXIT} after selected nodes that prunes all downstream nodes once a confidence signal, such as agreement among parallel agents, exceeds a threshold τ. A static workflow of depth D always pays $\sum_{j \le D} c_j$. Pruning makes the expected cost $\mathbb{E}\big[\sum_{j \le d} c_j\big]$ over a data-dependent exit depth $d \le D$, saving the most on easy queries. Representative methods: LayerSkip, DAAO, and FrugalGPT (Elhoushi et al., 2024; Su et al., 2026; Chen et al., 2023).

Strategy II: Heterogeneous node downscaling. Reassign the model M_i at each node according to its functional role, replacing M_{large} with M_{small} at low-entropy nodes (routing, formatting). Total cost becomes $\sum_i \text{FLOPs}(M_i)$ rather than $|N| \cdot \text{FLOPs}(M_{\text{large}})$, so right-sizing cuts cost, though role sensitivity is task-dependent. Because KV is specific to the model that computed it, a downscaled node also forfeits any prefix it shared with its neighbors. Representative methods: FrugalGPT, Hybrid-LLM, and X-MAS (Chen et al., 2023; Ding et al., 2024; Ye et al., 2025b).

Strategy III: Sparse expert decomposition. Replace one dense node with a router plus n specialized experts and activate only the $k \ll n$ relevant to a query. Active compute becomes $\propto k$ rather than $\propto n$, lifting the mixture-of-experts principle from layers to the workflow while leaving inactive expert subgraphs unexecuted. The price is the coverage that the unselected experts would have added. Representative methods: Switch Transformer, Mixtral, and MaAS (Fedus et al., 2022; Jiang et al., 2024; Zhang et al., 2025a).

Strategy IV: Router-based dynamic execution. A routing operator Op_{ROUTE} selects one execution path conditioned on the query or intermediate state. Per-query cost is then that of the chosen branch rather than the union of all branches, cutting computation on irrelevant paths at the price of runtime load imbalance. Representative systems: MaAS and other adaptive routing frameworks (Zhang et al., 2025a; Ong et al., 2025; Aggarwal et al., 2024).

Strategy V: Subgraph consolidation. Collapse a connected subgraph G_{sub} of tightly coupled agents into a single stronger node. This removes the internal edges E_{sub} and their inter-agent communication, replacing many short calls with one longer-context call and maximizing cache reuse, at the risk of losing role diversity. Representative methods: OneFlow and AgentArk (Xu et al., 2026; Luo et al., 2026b).

Interactions. The two families are not independent. Downscaling a node breaks the prefix it shared with same-model neighbors, and pruning removes the very calls whose shared prefix made caching attractive. Planning both under one cost model is what we mean by co-design.

4.3 Worked Example I: Multi-Agent Debate

To show that the catalogue *covers real systems*, we cast two structurally opposite frameworks as workflow graphs. The first is Multi-Agent Debate (Du et al., 2024). A debate of K agents over T rounds broadcasts the running transcript (a fan-out, Pattern II) and aggregates their answers at a per-round barrier (a fan-in, Pattern III), forming a diamond chain that alternates the two motifs. Within an agent, self-revision adds a reflection cycle (Pattern IV). Figure 2 maps each stage to its pattern and strategy.

The byte-identical broadcast prefix admits shared-prefix reuse (Pattern II) and the barrier admits critical-path scheduling (Pattern III), while consensus-based early exit (Strategy I), small-model early rounds (Strategy II), and consolidating easy queries (Strategy V) restructure the debate. Section 5 measures prefix reuse, early exit, and role-based downscaling on debate graphs of this form.

4.4 Worked Example II: MetaGPT

MetaGPT (Hong et al., 2024) is the structural opposite of debate. Five heterogeneous roles (product manager, architect, project manager, engineer, and QA) run in sequence rather than in parallel and exchange artifacts through a global message pool. Its node set mixes five computation nodes with tool nodes for search, debugging, and diagramming, its edges are dominated by compute–tool interleaving rather than broadcast, and the engineer node carries a self-loop in which an evaluation operator returns errors for repair (Figure 5, Appendix A).

The same catalogue nonetheless captures MetaGPT's salient structures. Its role chain is a sequential pipeline (Pattern I), which admits prefix reuse only if each role's prompt places the shared artifacts before its role-specific instructions, a layout choice that costs the algorithm nothing. Its tool calls are compute–tool overlap (Pattern VI), its repair loop is iterative reflection (Pattern IV), and its message pool is the blackboard of Pattern V. The rewrites transfer as well: a complexity gate fast-tracks simple requests past the design stages (Strategy I), a small model can take QA while a large one keeps architecture (Strategy II), and engineering can be dispatched to stack-specific experts (Strategy III). Section 5.3 measures the gate on a MetaGPT-style pipeline and expert dispatch on its own.

5 Experiments

We measure the serving benefit and accuracy effect of the patterns and strategies of Section 4 on real hardware, and test whether the cost model predicts which of them pay off end to end.

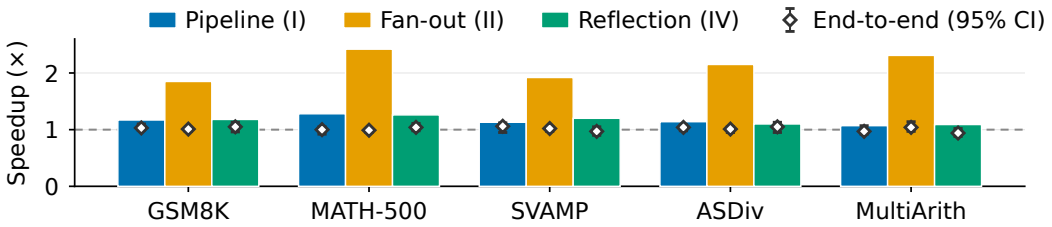

Figure 3: Prefix caching off vs. on for the three prefix-dominated static patterns. Bars are the aggregate TTFT speedup and markers the end-to-end speedup with 95% CIs, which straddle 1.0.

5.1 Experimental Setup

Models and serving. All experiments run on NVIDIA H100 GPUs with models served by vLLM (Kwon et al., 2023), version 0.18.1 for prefix reuse and 0.21.0 for the dynamic studies. We use Qwen3.5-9B throughout, plus Qwen3.5-4B as the small model for downscaling (Qwen Team, 2026). The two families run in different decoding regimes. The prefix-reuse study decodes in thinking mode (temperature 0.7, top-p 0.95, at most 32,768 tokens per call, chunked prefill), so its calls produce long reasoning traces that average 2.1k–11.6k tokens. Its baseline and cached arms run on the same GPU with a cold engine relaunch per arm, and the cached arm gates fan-out on agent 0's first token so that its siblings hit a warm prefix. The dynamic studies disable thinking and cap each call at 512 tokens at temperature 0.7, except the code study, which decodes greedily with a 768-token cap. They issue calls sequentially, so workflow latency is the sum of call latencies.

Workloads and protocol. The math studies use GSM8K, MATH-500, SVAMP, ASDiv, and MultiArith (Cobbe et al., 2021; Lightman et al., 2024; Patel et al., 2021; Miao et al., 2020; Roy & Roth, 2015), and the code study uses all 427 tasks of sanitized MBPP (Austin et al., 2021), scored by pass@1. For prefix reuse we run one chain at a time (seed 0) and analyze 130 paired chains per arm and benchmark after 20 warm-up chains. The chain patterns are a draft plus $T{=}4$ revision turns, and the fan-out pattern is a $K{=}3$ debate over 3 rounds answered by majority vote. TTFT speedup is $\sum \mathrm{TTFT(A0)} / \sum \mathrm{TTFT(A1)}$, end-to-end intervals are 10k-sample bootstrap CIs, and accuracy changes use an exact paired McNemar test. The debate studies run a $K{=}3$, $T{=}3$ debate (9 agent and 3 aggregator calls) on 100 tasks per benchmark (seed 42). Early exit measures agreement after each round as the size of the largest group of identical answers divided by K and stops once it reaches τ. Agreement takes the values $\{1/3, 2/3, 1\}$, so $\tau{=}1.0$ demands unanimity and $\tau{=}0.34$ a two-of-three majority. A run at $\tau{=}0.67$ also demands unanimity, because $2/3 < 0.67$, and serves as an independent replicate. Downscaling cost weights tokens by parameter count ($9.0\times$ large $+$ $4.0\times$ small). Appendix D details prompts and workflows.

5.2 Static Patterns: Prefix KV-Cache Reuse

Three of the six static patterns carry a reusable prefix: the pipeline (Pattern I), fan-out debate (Pattern II), and reflection (Pattern IV). Figure 3 reports a 1.07–2.42× aggregate TTFT speedup, concentrated in fan-out (1.85–2.42×), whose $K{=}3$ agents share a byte-identical prefix prefilled once. The chain patterns reuse a shorter cross-turn prefix (1.07–1.28×). Hit rates rank fan-out first (0.51–0.55 of prompt tokens, against 0.32–0.50 for the chains), as Pattern II predicts, because the shared prefix is read K times per round. Temperature-0.7 sampling lets the arms diverge per question, yet accuracy moves by at most 0.008, one question in 130, and McNemar fails to reject ($p{=}1.00$), which bounds the effect without establishing equivalence.

End to end, the same runs give 0.94–1.06×, and every 95% CI covers 1.0. The cost model predicts this parity. Decode takes 95.6–99.8% of latency, so the prefill share f in Equation 3 is at most 4.4%. Substituting each cell's prefill share and TTFT speedup caps the end-to-end gain at 1.0003–1.027×. No measured CI lies above the cap, and 14 of the 15 contain it, the exception falling just below (Table 3). Prefix reuse therefore saves prefill compute but not request latency in this decode-bound regime. Its payoff grows with the prefill share, as in long shared contexts with short answers, so an optimizer must read the phase profile, not just the graph.

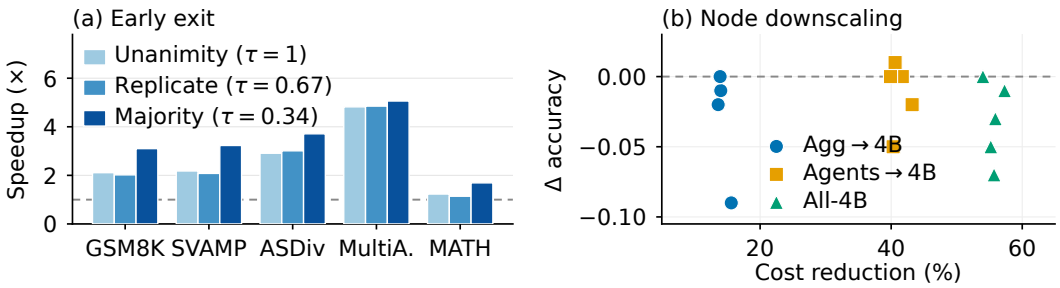

Figure 4: Dynamic strategies on the K=3, T=3 debate. (a) Early-exit speedup over the full debate under each stopping rule. (b) Node downscaling, cost reduction vs. accuracy change, one marker per benchmark. No arm dominates on all five benchmarks.

Table 1: Early exit (Strategy I) on the K=3, T=3 debate (100 tasks per benchmark). Full is the accuracy of the full debate, Δ the accuracy change under early exit, Speedup the latency ratio, and Tok. the fraction of tokens saved.

		Unanimity (τ=1.0)			**Majority** (τ=0.34)		
Benchmark	**Full**	Δ	Speedup	Tok.	Δ	Speedup	Tok.
GSM8K	0.92	+0.03	2.11×	59.5%	+0.02	3.10×	78.1%
SVAMP	0.92	+0.01	2.18×	60.5%	+0.00	3.23×	78.5%
ASDiv	0.95	−0.01	2.91×	73.7%	+0.01	3.71×	82.3%
MultiArith	1.00	−0.01	4.82×	87.6%	+0.00	5.06×	88.7%
MATH-500	0.45	−0.02	1.23×	22.5%	−0.05	1.69×	53.5%

5.3 Dynamic Strategies

Dynamic strategies change what is computed, so both cost and accuracy can move, and they remove decode work along with prefill, so their savings reach end-to-end latency. Two independent runs of the full debate agree within 0.02 on every benchmark yet disagree on 20 individual MATH-500 questions, and no early-exit accuracy change is significant under a paired McNemar test ($p \geq 0.125$), so changes of a few points are within run-to-run noise.

Adaptive pruning via early exit. Early exit (Strategy I) stops the debate after the first round whose agreement reaches τ (Table 1, Figure 4a). On the four easier benchmarks it is nearly free. The majority rule removes 61–67% of LLM calls and 78–89% of tokens for a 3.1–5.1× speedup while accuracy stays within 0.02 of the full debate, and the stricter unanimity rule still runs 2.1–4.8× faster. On MATH-500 agents agree less often, so both rules retain more rounds (43% of calls removed under majority, 19% under unanimity) for a 1.2–1.7× speedup, and accuracy falls by 0.02–0.05 from 0.45 ($p \geq 0.27$). The unanimity replicate matches the first run within 0.05 in accuracy and 5 points in call savings, with the largest gaps on MATH-500 (Table 4). The workflow thus answers easy questions quickly and deliberates longer on hard ones, where stopping early is most likely to cost accuracy.

Heterogeneous node downscaling. Node downscaling (Strategy II) moves selected debate nodes to Qwen3.5-4B (Figure 4b, Table 5). Placing the small model on the three aggregators (Agg→4B) saves only 13.6–15.6% of cost yet loses 0.09 on MATH-500 (p=0.049, the only significant change), where the aggregator writes the final answer. Placing it on the nine agents (Agents→4B) saves 39.9–43.2% for a 1.34–1.51× speedup, within 0.02 of All-9B on the four easier benchmarks and 0.05 below it on MATH-500. All-4B saves 54.0–57.3% for a 1.35–1.59× speedup and matches or beats Agents→4B on MultiArith and ASDiv but falls 0.05–0.07 below All-9B on GSM8K and MATH-500. No arm wins everywhere. Where the small model goes matters as much as how many nodes carry it.

Code workflows: pipeline pruning and sparse experts. Table 2 evaluates the MetaGPT-style pipeline of Section 4.4 and sparse expert dispatch on MBPP against a single direct call. The five-role

Table 2: Code workflows on sanitized MBPP (427 tasks, greedy decoding). Calls, tokens, and latency are per-task means, and Tok./pass divides tokens by pass@1.

Workflow	Calls	Tokens	Latency (s)	pass@1	Tok./pass
Single direct call	1.00	168	0.44	0.775	217
MetaGPT-style pipeline (5 roles)	5.00	4,809	11.04	0.658	7,308
+ complexity gate (Strategy I)	3.18	1,661	4.23	0.703	2,364
Five experts + selector (dense)	6.00	2,123	3.16	0.775	2,739
one routed expert (Strategy III)	1.00	314	0.55	0.747	421

pipeline reaches only 0.658 pass@1 at 5 calls and 11.0 s per task, below the direct call's 0.775 at one call and 0.44 s. A zero-cost complexity gate (Strategy I) sends short tasks without hard keywords from the product manager straight to the engineer. It skips the design stages on 388 of 427 tasks, cutting tokens $2.9\times$ and latency $2.6\times$ while raising pass@1 to 0.703. Sparse dispatch (Strategy III) routes each task by a keyword rule to one of five prompt-specialized experts instead of running all five plus a selector call. It cuts calls $6.0\times$, tokens $6.8\times$, and latency $5.8\times$ for 2.8 points of pass@1 (0.747 vs. 0.775). The dense ensemble, however, only matches the direct call, and no multi-agent configuration beats it. The cheapest single-call configuration is thus the baseline every rewrite must beat, a check that accuracy-only evaluation of MAS rarely runs.

6 Discussion: Toward Graph Auto-Optimization

Our optimizations are hand-designed. Once a workflow is visible as a typed graph, the natural next step is to optimize it *automatically* under explicit serving objectives while preserving task quality. Relaxing each control edge into a probability $p(e_{i\to j} \mid x, s)$, as GPTSwarm does for connectivity (Zhuge et al., 2024), turns the graph into a policy π_θ over executions, and auto-optimization into minimizing $\mathbb{E}_{\pi_\theta(G)}[\, C_{\text{sys}}(G,x) + \lambda\, C_{\text{qual}}(G,x)\,]$ with C_{sys} priced by Equation 2. The strategies of Section 4.2 then mirror network compression: dropping redundant agents is pruning, moving a node to a smaller model is like shrinking a layer, and collapsing a subgraph is distillation. The cost term must be faithful. Token counts mispredict multi-agent cost, because prefill and decode tokens take very different time (Section 5.2), and deployment terms such as KV residency, prefill bursts at fan-out, barrier slack at fan-in, and memory held during tool calls are non-additive and scheduling-dependent. An auto-optimizer must co-optimize topology, serving policy, and placement (Appendix C).

Limitations. Our measurements cover one model family (Qwen3.5 at 4B and 9B), math and code benchmarks, and single-request serving, without throughput, multi-tenant contention, or real tool calls, so Patterns III, V, and VI and Strategies IV and V rest on the graph analysis and prior systems. With 100 tasks per benchmark, few-point accuracy changes in the dynamic studies are within noise.

7 Conclusion

Multi-agent LLM serving is a full-stack problem. Typed workflow graphs with a phase-aware cost model name what a runtime can exploit, what a designer can rewrite, and when each pays off: prefix reuse is capped by the prefill share, rewrites that remove decode work pay off at a measurable accuracy cost, and some multi-agent structure buys nothing over a single call. We hope algorithm designers will report the serving cost of their workflows next to accuracy, and that serving systems will take workflow graphs, not isolated prompts, as their unit of work.

AI use statement

In this work, we used generative AI tools for the following tasks with required disclosure. An AI assistant proposed the phase-aware cost model and its Amdahl bound (Section 3), which the authors reviewed and adopted, and it helped interpret the recorded results by cross-checking every reported number against the authors' experiment logs and code and by computing the paired significance tests

reported in Section 5.3. That check found a sign error in an earlier version of the early-exit table and showed that two of the tested thresholds implement the same stopping rule, both of which we now report. The remaining tasks with required disclosure are not applicable to this work. Additionally, we used generative AI tools for tasks with recommended disclosure: editing text for readability, drafting revisions of author-written sections, converting the manuscript to the ICLR template, writing the scripts that render the recorded results as figures and tables, adjusting figure layout, and finding, verifying, and formatting references. We have reviewed all AI-assisted work, checked every reported number against the raw result files, and verified every reference against its publisher's record. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

Ethics statement

This work studies the serving efficiency of multi-agent LLM workflows on public benchmarks (GSM8K, MATH-500, SVAMP, ASDiv, MultiArith, and MBPP). It involves no human subjects and no personal or sensitive data. Making multi-agent systems cheaper to serve reduces their energy use and cost.

Reproducibility statement

Section 5.1 and Appendix D specify the hardware, serving engine versions, models, decoding settings, sample sizes, seeds, stopping rules, and metrics of every study. Appendix D lists the complete per-benchmark results, including the absolute accuracies behind every reported change, the end-to-end confidence intervals, the decode shares, and the Amdahl bounds of Section 5.2. Every table and figure in Section 5 is generated from the recorded result files by scripts, so the reported numbers can be traced to the raw measurements.

References

Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. InferCept: Efficient intercept support for augmented large language model inference. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 81–95. PMLR, 2024.

Pranjal Aggarwal, Aman Madaan, Ankit Anand, Srividya Pranavi Potharaju, Swaroop Mishra, Pei Zhou, Aditya Gupta, Dheeraj Rajagopal, Karthik Kappaganthu, Yiming Yang, Shyam Upadhyay, Manaal Faruqui, and Mausam. AutoMix: Automatically mixing language models. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, 2024.

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. 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. USENIX Association, 2024. URL https://www.usenix.org/conference/osdi24/presentation/agrawal.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.

Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. Small language models are the future of agentic ai. arXiv preprint arXiv:2506.02153, 2025.

Lingjiao Chen, Matei Zaharia, and James Zou. Frugalgpt: How to use large language models while reducing cost and improving performance, 2023. URL https://arxiv.org/abs/2305.05176.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021.

DeepSeek-AI. Deepseek-v3 technical report, 2024. URL https://arxiv.org/abs/2412.19437.

Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Ruhle, Laks V. S. Lakshmanan, and Ahmed Hassan Awadallah. Hybrid LLM: Cost-efficient and quality-aware query routing. In *International Conference on Learning Representations (ICLR)*, 2024.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 11733–11763. PMLR, 2024.

Mostafa Elhoushi, Akshat Shrivastava, Diana Liskovich, Basil Hosmer, Bram Wasti, Liangzhen Lai, Anas Mahmoud, Bilge Acun, Saurabh Agarwal, Ahmed Roman, Ahmed Aly, Beidi Chen, and Carole-Jean Wu. Layerskip: Enabling early exit inference and self-speculative decoding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12622–12642, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.681. URL https://aclanthology.org/2024.acl-long.681/.

Siqi Fan, Xin Jiang, Xiang Li, Xuying Meng, Peng Han, Shuo Shang, Aixin Sun, Yequan Wang, and Zhongyuan Wang. Not all layers of LLMs are necessary during inference. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2025*, pp. 5083–5091. ijcai.org, 2025. doi: 10.24963/ijcai.2025/566. AdaInfer; also arXiv:2403.02181, 2024.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022. URL http://jmlr.org/papers/v23/21-0998.html.

Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. ServerlessLLM: Low-latency serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 135–153, Santa Clara, CA, 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/fu.

Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. MegaBlocks: Efficient sparse training with mixture-of-experts. In *Proceedings of Machine Learning and Systems (MLSys)*, volume 5, pp. 288–304, 2023.

Shubham Gandhi, Manasi Patwardhan, Lovekesh Vig, and Gautam Shroff. BudgetMLAgent: A cost-effective LLM multi-agent system for automating machine learning tasks. In *Proceedings of the 4th International Conference on AI-ML Systems (AIMLSystems '24)*, pp. 1–9, Baton Rouge, LA, USA, 2024. Association for Computing Machinery. doi: 10.1145/3703412.3703416.

In Gim, Zhiyao Ma, Seung-seob Lee, and Lin Zhong. Pie: A programmable serving system for emerging llm applications. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP '25)*. Association for Computing Machinery, 2025. doi: 10.1145/3731569.3764814.

Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. MiniLLM: Knowledge distillation of large language models. In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2306.08543.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2308.00352.

Cheng-Yu Hsieh, Chun-Liang Li, Chih-kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alex Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 8003–8017, Toronto, Canada, July 2023.

Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-acl.507. URL https://aclanthology.org/2023.findings-acl.507/.

Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts. arXiv preprint arXiv:2401.04088, 2024.

Yubin Kim, Ken Gu, Chanwoo Park, Chunjong Park, Samuel Schmidgall, A. Ali Heydari, Yao Yan, Zhihan Zhang, Yuchen Zhuang, Yun Liu, Mark Malhotra, Paul Pu Liang, Hae Won Park, Yuzhe Yang, Xuhai Xu, Yilun Du, Shwetak Patel, Tim Althoff, Daniel McDuff, and Xin Liu. Towards a science of scaling agent systems. arXiv preprint arXiv:2512.08296, 2025. URL https://arxiv.org/abs/2512.08296.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*, pp. 611–626. Association for Computing Machinery, 2023. doi: 10.1145/3600006.3613165.

LangChain. LangGraph: Build stateful, multi-actor applications with LLMs. https://langchain-ai.github.io/langgraph/, 2024. Official documentation, LangChain. Accessed 2024. Docs have since moved to https://docs.langchain.com/oss/python/langgraph/.

Weizhen Li, Jianbo Lin, Zhuosong Jiang, Jingyi Cao, Xinpeng Liu, Jiayu Zhang, Zhenqiang Huang, Qianben Chen, Weichen Sun, Qiexiang Wang, Hongxuan Lu, Tianrui Qin, Chenghao Zhu, Yi Yao, Shuying Fan, Xiaowan Li, Tiannan Wang, Pai Liu, King Zhu, He Zhu, Dingfeng Shi, Piaohong Wang, Yeyi Guan, Xiangru Tang, Minghao Liu, Yuchen Eleanor Jiang, Jian Yang, Jiaheng Liu, Ge Zhang, and Wangchunshu Zhou. Chain-of-agents: End-to-end agent foundation models via multi-agent distillation and agentic rl. arXiv preprint arXiv:2508.13167, 2025. URL https://arxiv.org/abs/2508.13167.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024.

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

Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E. Gonzalez, and Ion Stoica. Agentix: An efficient serving engine for LLM agents as general programs. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pp. 2443–2459, Renton, WA, 2026a. USENIX Association.

Yinyi Luo, Yiqiao Jin, Weichen Yu, Mengqi Zhang, Srijan Kumar, Xiaoxiao Li, Weijie Xu, Xin Chen, and Jindong Wang. Agentark: Distilling multi-agent intelligence into a single llm agent. arXiv preprint arXiv:2602.03955, 2026b.

Yacine Majdoub, Eya Ben Charrada, and Haifa Touati. Towards adaptive software agents for debugging. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, FSE Companion 2025, Trondheim, Norway, June 23-28, 2025*, pp. 636–640. ACM, 2025. doi: 10.1145/3696630.3728514.

Shen-yun Miao, Chao-Chun Liang, and Keh-Yih Su. A diverse corpus for evaluating and developing English math word problem solvers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 975–984, 2020.

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M Waleed Kadous, and Ion Stoica. RouteLLM: Learning to route LLMs from preference data. In *International Conference on Learning Representations (ICLR)*, 2025.

Zaifeng Pan, Ajjkumar Patel, Zhengding Hu, Yipeng Shen, 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 (NeurIPS)*, 2025.

Arkil Patel, Satwik Bhattamishra, and Navin Goyal. Are NLP models really able to solve simple math word problems? In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pp. 2080–2094, 2021.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–132. IEEE, 2024. doi: 10.1109/ISCA59077.2024.00019.

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

Subhro Roy and Dan Roth. Solving general arithmetic word problems. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1743–1752, 2015.

Junyi Shen, Noppanat Wadlom, and Yao Lu. Batch query processing and optimization for agentic workflows, 2025. URL https://arxiv.org/abs/2509.02121. arXiv:2509.02121.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems 36 (NeurIPS)*, 2023.

Jinwei Su, Qizhen Lan, Yinghui Xia, Lifan Sun, Weiyou Tian, Tianyu Shi, Xinyuan Song, Lewei He, and Jingsong Yang. Difficulty-aware agentic orchestration for query-specific multi-agent workflows. In *Proceedings of the ACM Web Conference 2026 (WWW '26)*, Dubai, United Arab Emirates, 2026. ACM. doi: 10.1145/3774904.3792240.

Xin Tan, Yimin Jiang, Yitao Yang, and Hong Xu. Towards end-to-end optimization of LLM-based applications with Ayo. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25)*, pp. 1302–1316. ACM, 2025. doi: 10.1145/3676641.3716278.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling (COLM)*, 2024. arXiv:2308.08155.

Jiawei Xu, Arief Koesdwiady, Sisong Bei, Yan Han, Baixiang Huang, Dakuo Wang, Yutong Chen, Zheshen Wang, Peihao Wang, Pan Li, and Ying Ding. Rethinking the value of multi-agent workflow: A strong single agent baseline, 2026. arXiv:2601.12307.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems 36 (NeurIPS)*, 2023a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations (ICLR)*, 2023b.

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, pp. 17882–17928, 2025a.

Rui Ye, Xiangrui Liu, Qimin Wu, Xianghe Pang, Zhenfei Yin, Lei Bai, and Siheng Chen. X-mas: Towards building multi-agent systems with heterogeneous llms, 2025b. URL https://arxiv.org/abs/2505.16997.

Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. Multi-agent architecture search via agentic supernet. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, 2025a. arXiv:2502.04180.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025b. arXiv:2410.10762.

Meicong Zhang, Tiancheng Su, and Guoxiu He. Eliminating agentic workflow for introduction generation with parametric stage tokens, 2025c. arXiv:2601.09728.

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 37 (NeurIPS 2024)*, 2024.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 193–210, Santa Clara, CA, 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language agents as optimizable graphs. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62743–62767. PMLR, 2024.

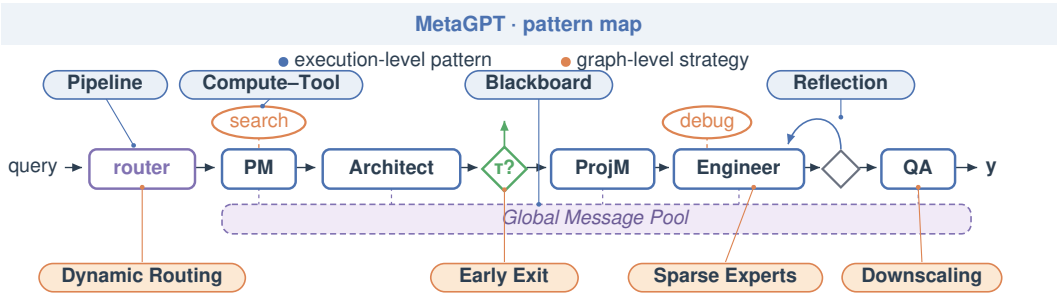

Figure 5: MetaGPT as a pattern map. A pipeline of five specialized roles communicates through a global message pool, annotated with the execution-level pattern and graph-level strategy that apply at each stage.

A Extended Case Study: MetaGPT

Section 4.4 summarizes MetaGPT as a worked example. This appendix gives its full graph formalization and the optimizations it admits.

MetaGPT represents a software-engineering-inspired coordination paradigm (Hong et al., 2024). Unlike the homogeneous reasoning of Multi-Agent Debate, it introduces five specialized roles (product manager, architect, project manager, engineer, and QA engineer). Standard operating procedures coordinate these roles into a structured pipeline in which agents refine shared artifacts through a publish–subscribe mechanism over a global message pool.

Graph formalization. We formalize the MetaGPT workflow as a graph $G_{\text{MetaGPT}} = (N, E, Op)$. The node set is heterogeneous. The computation nodes $N_{\text{compute}} = \{N_{\text{PM}}, N_{\text{Arch}}, N_{\text{ProjM}}, N_{\text{Eng}}, N_{\text{QA}}\}$ are role-specialized agents of the software development lifecycle, and the tool nodes $N_{\text{tool}} = \{N_{\text{search}}, N_{\text{debug}}, N_{\text{diagram}}\}$ provide deterministic external functions such as web search, execution debugging, and system visualization, so $N = N_{\text{compute}} \cup N_{\text{tool}}$. Besides the sequential edges that carry artifacts from role to role, two edge classes, $E_{\text{int}} \cup E_{\text{ref}} \subseteq E$, govern the behavior within a role.

- **Compute–tool interleaving edges (E_{int})** connect computation nodes to tool nodes. For example, $\langle N_{\text{PM}}, N_{\text{search}} \rangle \in E_{\text{int}}$ lets the product manager retrieve web information.
- **Iterative reflection edges (E_{ref})** close a loop inside the engineer node. N_{Eng} generates code that an execution operator $Op_{\text{eval}} \in Op$ evaluates, and if the feedback reports errors or violated constraints, a reflection edge $\langle Op_{\text{eval}}, N_{\text{Eng}} \rangle \in E_{\text{ref}}$ routes it back to the agent for repair.

As Figure 5 illustrates, this formalization characterizes MetaGPT as an artifact-driven pipeline whose execution follows structured dependencies between heterogeneous nodes. At the macro level it is a role-specialized sequential pipeline (Pattern I). Within a role it interleaves computation with tools (Pattern VI), and the engineer node adds iterative reflection (Pattern IV). These structures admit the optimizations of Section 4 directly.

Execution-level optimizations. The pipeline's structural determinism and role specialization expose three semantics-preserving optimizations.

- **Prefix-first prompt layout for the role chain (Pattern I).** Successive roles consume the same growing set of artifacts but prepend different role instructions, so their prompts share no token-identical prefix and prefix caching finds nothing to reuse. Placing the shared artifacts first and the role instructions last turns the chain into a growing prefix that each role prefills only incrementally. The layout change is invisible to the algorithm.
- **Proactive memory yielding during tool latency (Pattern VI).** The interleaving edges E_{int} idle the GPU during long tool calls such as N_{search} or complex N_{debug} sessions, while the calling agent's KV stays resident. Since a traversal of E_{int} is visible in G, an intercept-aware runtime such as InferCept (Abhyankar et al., 2024) can decide at that moment whether to

keep, discard, or swap the KV to host memory, turning an unpredictable wait into memory that concurrent requests can use.

- **Context pinning for debugging cycles (Pattern IV).** Inside N_{Eng}, the reflection edges E_{ref} form a high-locality loop. The error feedback changes every iteration, but the task instructions and most of the code do not. A programmable cache controller such as Pie (Gim et al., 2025) can pin the KV of this invariant prefix so that each repair round prefills only the new feedback tokens.

Graph-level optimizations. Execution-level techniques speed up the existing pipeline. Graph-level optimizations instead rewrite the MetaGPT topology to remove redundancy inherent in its development process.

- **Adaptive pipeline pruning via task-complexity estimation (Strategy I).** The full procedure is often over-engineered for simple requests such as "write a Python script that sorts a CSV file." A conditional operator Op_{EXIT} after requirement analysis can assess task complexity and route simple requirements directly to N_{Eng}, skipping the design stages that add little to trivial implementations. Section 5.3 measures such a gate on a MetaGPT-style code pipeline, where it skips the design stages on 388 of 427 MBPP tasks, cuts tokens $2.9\times$, and raises pass@1 from 0.658 to 0.703.
- **Heterogeneous role–model alignment (Strategy II).** MetaGPT's heterogeneity lives in its personas, yet it typically runs every role on the same frontier model. Node-level downscaling instead assigns each N_i a backend matched to its role. The architect's system-wide trade-offs call for a large model, whereas QA and routine engineering can run on smaller or code-specialized models, following evidence that right-sized models suffice for many agentic subtasks (Belcak et al., 2025).
- **Sparse expert dispatch for multi-stack projects (Strategy III).** Large projects span several technology stacks, for example a React frontend and a Go backend. Rather than one N_{Eng} node, the engineering phase can become a sparse expert subgraph in which a router reads the project manager's task list and activates only the relevant experts (frontend, backend, or DevOps). This prevents context pollution within a single engineer and lets stack-specialized experts run concurrently.

B Extended Discussion of Graph-Level Optimizations

This appendix adds technical detail and literature context for the dynamic strategies of Section 4.2.

B.1 Adaptive Subgraph Pruning

Adaptive subgraph pruning is motivated by a simple empirical fact: not all queries require the full depth of a multi-agent workflow. Query difficulty is highly skewed in realistic deployments, and early-stage agents can handle many instances correctly without engaging the full pipeline of drafting, critique, refinement, and verification. Running every query through the complete graph therefore overcomputes systematically.

Pruning pays off in three ways. First, it directly reduces the number of LLM invocations. If a pipeline of depth k exits after stage j, the system avoids $k - j$ calls. Second, the token savings often exceed the call savings, because later agents usually consume the accumulated outputs of earlier ones, so skipping them avoids both their generation and their growing prefill. Third, pruning improves memory efficiency. KV caches for skipped nodes are never allocated, and completed caches can be freed earlier because no downstream consumer remains.

Several lines of prior work support this pattern. At the model level, LayerSkip (Elhoushi et al., 2024) and AdaInfer (Fan et al., 2025) show that inputs of different difficulty can profitably use different effective depths. At the routing level, FrugalGPT (Chen et al., 2023), RouteLLM (Ong et al., 2025), AutoMix (Aggarwal et al., 2024), and Hybrid LLM (Ding et al., 2024) apply adaptive escalation across models. At the workflow level, DAAO (Su et al., 2026) builds simpler workflows for easy queries and richer ones for hard queries, and MaAS (Zhang et al., 2025a) samples query-dependent subnetworks from an agentic supernet. STIG (Zhang et al., 2025c) shows that in structured settings the

strongest form of pruning can collapse the entire workflow into a single call, and adaptive debugging systems (Majdoub et al., 2025) likewise find that easy instances need minimal agentic depth whereas difficult ones benefit from more collaboration.

B.2 Heterogeneous Node Downscaling

Heterogeneous node downscaling addresses a recurring mismatch in multi-agent systems. Nodes often perform narrow functions, yet many frameworks instantiate every node with the same large frontier model. This homogeneous design over-provisions compute for roles such as summarization, formatting, verification, parsing, or constrained tool invocation.

The systems benefits are direct. Replacing a large model with a smaller one at even a single node substantially reduces the FLOPs per invocation. Smaller models also reduce memory footprint and KV-cache growth, which improves concurrency and lowers infrastructure cost. Because small language models are easy to fine-tune or adapt with lightweight methods such as LoRA, the workflow graph also offers a natural path toward progressive specialization: once a node's role has been observed repeatedly, the system can replace a generalist LLM with a right-sized specialist. The cost is a cache conflict. KV computed by one model cannot serve another, so a downscaled node loses any prefix it shared with its neighbors, and the planner must weigh the cheaper tokens against the lost reuse.

The literature supports this view from several angles. Distilling Step-by-Step (Hsieh et al., 2023) and MiniLLM (Gu et al., 2024) establish that task-relevant capabilities transfer from large to small models. Belcak et al. (2025) argue that many agentic tasks are better matched to small language models than to frontier LLMs and propose an explicit LLM-to-SLM conversion process based on usage clustering and iterative specialization. X-MAS (Ye et al., 2025b) finds that no single model dominates across all functions and domains, which is direct evidence for heterogeneous assignment. DAAO (Su et al., 2026) further operationalizes node-level routing to different models, and BudgetMLAgent (Gandhi et al., 2024) shows that replacing most worker calls with cheaper models can greatly reduce cost while maintaining or improving the success rate. From a serving perspective, the strategy aligns with disaggregated systems such as Splitwise (Patel et al., 2024) and DistServe (Zhong et al., 2024), which already separate phases or resource profiles across hardware tiers.

B.3 Sparse Expert Decomposition

Sparse expert decomposition generalizes the mixture-of-experts principle from model internals to workflow structure. Instead of one dense agent for all query types, the system replaces that node with a router and a collection of smaller experts. Only a small subset of experts is activated for each query, after which their outputs are aggregated and passed forward.

This yields several systems advantages. First, it creates sparse activation at the agent level, so the active compute per request depends on the selected experts rather than on the full capacity of a generalist node. Second, it encourages focused specialization, since each expert can target a narrow subtask such as numerical reasoning, SQL generation, citation formatting, or data extraction. Third, because experts are smaller, they can be distributed flexibly across devices or replicated according to demand, which connects the workflow abstraction to hardware-aware scheduling.

The idea is grounded in successful model-level MoE systems such as Switch Transformer (Fedus et al., 2022), Mixtral (Jiang et al., 2024), and DeepSeek-V3 (DeepSeek-AI, 2024), all of which show that sparse activation provides large total capacity at much lower active compute. At the workflow level, DAAO (Su et al., 2026) can be read as a router that selects among heterogeneous operator–model pairs, and MaAS (Zhang et al., 2025a) is an especially direct analogue that samples a query-dependent subnetwork from an agentic supernet. From a systems standpoint, the router is then both a capability selector and a placement decision, much like expert parallelism in MegaBlocks (Gale et al., 2023).

B.4 Subgraph Consolidation

Subgraph consolidation is the inverse of pruning and the dual of expert decomposition. Instead of expanding or selectively skipping structure, it collapses an existing multi-agent region into a single node executed by a stronger model. It is motivated by the growing observation that many multi-agent

workflows, especially homogeneous ones, incur coordination and communication overhead without clear gains in quality. Our code study offers one such case, since a single direct call matches a five-expert ensemble (Section 5.3).

The benefits are threefold. First, consolidation eliminates internal communication edges and with them serialization, hand-off, and repeated prefill of the same messages. Second, it avoids context fragmentation, since the task is processed in one coherent context rather than split across several agent windows. Third, it mitigates error cascades, because intermediate outputs no longer serve as brittle interfaces between stages.

Several recent works support this pattern. OneFlow (Xu et al., 2026) shows that homogeneous multi-agent workflows can often be simulated by a single LLM through multi-turn execution, with the added benefit of KV-cache reuse. AgentArk (Luo et al., 2026b) distills debate-style multi-agent behavior into a single model's parameters, shifting orchestration cost from inference time to training time. Chain-of-Agents (Li et al., 2025) similarly distills multi-agent trajectories into agent foundation models that can outperform the original systems. STIG (Zhang et al., 2025c) is a domain-specific example in which a single model with stage tokens replaces a multi-stage academic-writing workflow. In serving terms, consolidation replaces many short calls with one long-context call, which removes repeated prefill but lengthens attention over the single context.

Consolidation is not universally appropriate. It works best when the subgraph is relatively homogeneous, when the relevant context fits in one model's window, and when one model can subsume the constituent functions. When these conditions fail, the decomposed architecture may remain preferable. This motivates conditional consolidation, in which the runtime profiles subgraphs and consolidates only those whose expected quality loss stays below a chosen threshold.

C Extended Discussion: Toward Graph Auto-Optimization

This appendix expands the outlook of Section 6.

From hand-designed to learned graphs. Relaxing the workflow into a **probabilistic execution model** makes it optimizable. Each control edge carries a probability $p(e_{i\to j} \mid x, s)$, generalizing GPTSwarm's learned connectivity (Zhuge et al., 2024). Given the per-node cost of Equation 2, the runtime objective becomes minimizing $\mathbb{E}_{\pi_\theta(G)}[\, C_{\text{sys}}(G,x) + \lambda\, C_{\text{qual}}(G,x)\,]$, where π_θ parameterizes the edge probabilities and C_{qual} encodes quality constraints. Hard operators relax into differentiable gates trained with straight-through or policy-gradient estimators. Probabilistic edges also let the runtime prefetch KV in proportion to transition probability, as KVFlow does along the deterministic next step (Pan et al., 2025). This invites a neural-network view in which each node is a parameterized "neuron" and the dynamic strategies mirror network compression. *Subgraph dropout* disables redundant parallel agents to expose pruning candidates (Strategy I), *node-level downshifting* replaces large backends with cheaper ones (Strategy II), and *subgraph distillation* collapses a multi-node region into one stronger node (Strategy V). Topology search then parallels architecture search, as in AFlow-style workflow generation (Zhang et al., 2025b), but with serving-aware cost terms as first-class constraints. Collecting edge gates, node backends, operator thresholds, and system-policy knobs into θ turns auto-optimization into a constrained program that minimizes serving cost subject to quality constraints from verifiers or downstream evaluations.

Serving-aware objectives and a compiler–runtime stack. Auto-optimization needs objectives faithful to deployment. The graph enables richer terms than token counts: KV-residency cost (peak and integral of GPU KV, which bounds batch size), a prefill-shock penalty for concurrent fan-out prefills, barrier-induced idle time from fan-in stragglers, and tool-induced dead weight from GPU memory held during tool calls. Because these terms are non-additive and scheduling-dependent, topology, policy, and placement must be **co-optimized**. We envision a compiler–runtime stack paralleling deep-learning compilers: a typed graph IR with state-growth rules, correctness-by-construction rewrites guarded by operator contracts and fast validators in a "verify-then-apply" loop, and profiling that turns execution traces into a training set for learned routers and gates. Open challenges include non-stationary evaluation, heavy-tailed tool latency, guardrails against silently broken corner cases, and cluster-level fairness, which motivate shared benchmarks of typed workflow graphs with traces and evaluators.

Table 3: Full prefix-reuse results (n=130 per cell). Acc. is accuracy with caching off (A0) and on (A1), TTFT the aggregate time-to-first-token speedup, E2E the end-to-end speedup with its bootstrap 95% CI, Decode the decode share of latency, Bound the Amdahl ceiling of Equation 3 computed from the TTFT speedup and the prefill share, and Hit the fraction of prompt tokens served from cache.

Benchmark	Acc. A0	Acc. A1	TTFT	E2E [95% CI]	Decode	Bound	Hit
Pipeline (I)							
GSM8K	0.977	0.977	1.17×	1.03× [0.99, 1.10]	99.7%	1.000×	0.43
MATH-500	0.892	0.885	1.28×	1.00× [0.92, 1.05]	99.5%	1.001×	0.49
SVAMP	0.938	0.946	1.13×	1.06× [0.95, 1.12]	99.7%	1.000×	0.38
ASDiv	0.985	0.992	1.14×	1.04× [0.98, 1.09]	99.1%	1.001×	0.36
MultiArith	0.962	0.969	1.07×	0.97× [0.92, 1.07]	99.1%	1.001×	0.37
Fan-out (II)							
GSM8K	0.969	0.969	1.85×	1.01× [0.97, 1.04]	98.9%	1.005×	0.51
MATH-500	0.892	0.885	2.42×	0.99× [0.95, 1.01]	95.6%	1.027×	0.53
SVAMP	0.946	0.946	1.92×	1.02× [0.97, 1.05]	99.0%	1.005×	0.55
ASDiv	0.992	0.992	2.15×	1.01× [0.95, 1.05]	98.7%	1.007×	0.55
MultiArith	0.962	0.962	2.31×	1.04× [0.97, 1.14]	98.8%	1.007×	0.55
Reflection (IV)							
GSM8K	0.962	0.969	1.18×	1.05× [0.96, 1.13]	99.7%	1.000×	0.42
MATH-500	0.877	0.869	1.26×	1.04× [0.98, 1.11]	99.8%	1.000×	0.50
SVAMP	0.946	0.946	1.20×	0.97× [0.90, 1.05]	99.3%	1.001×	0.38
ASDiv	0.992	0.992	1.10×	1.05× [0.95, 1.13]	99.6%	1.000×	0.32
MultiArith	0.962	0.969	1.09×	0.94× [0.88, 1.02]	99.4%	1.000×	0.34

D Experimental Details and Full Results

D.1 Prefix KV-Cache Reuse

The engine runs with `--max-model-len` 40960 and chunked prefill in latency mode (concurrency 1, seed 0). Each arm serves 150 chains per benchmark, of which the first 20 are discarded as server warm-up, and the remaining 130 are paired by question across arms. The pipeline chain keeps one growing conversation per question: an initial draft followed by T=4 refinement turns that ask the model to review the solution so far and produce an improved complete solution, so turn t+1's prompt extends turn t's conversation. The reflection chain has the same mechanics with self-critique wording that asks the model to find and correct mistakes in its previous solution. In the fan-out debate, the prompts of rounds after the first share a common prefix consisting of the question and the broadcast transcript tails (1,200 characters per agent), and the answer is the majority vote over the three agents' final-round answers. Accuracy is checked numerically or with a math verifier on the final answer. Calls are long: mean tokens per call range from 2.1k (reflection on ASDiv) to 11.6k (fan-out on MATH-500), and up to 13.9% of MATH-500 calls reach the 32,768-token cap. Table 3 lists the accuracy of both arms, the end-to-end intervals, the decode share, and the Amdahl bound of Equation 3 for every cell.

D.2 Debate Studies: Early Exit and Node Downscaling

The debate studies sample 100 tasks per benchmark with seed 42. Each debate round issues three agent calls and one aggregator call. In the first round each agent answers the question alone, and in later rounds it sees all of its own previous solutions and its two peers' latest solutions. The aggregator reads the round's three solutions and writes the round's answer, and the final answer is that of the last aggregator. Early exit computes agreement over the three agents' extracted answers after each round. Tables 4 and 5 list the absolute accuracy of every run. The full-debate baselines of the two studies are independent runs with the same workflow and decoding configuration. They agree within 0.02 on every benchmark but disagree on individual questions, most on MATH-500 (20 of 100). Under an

Table 4: Full early-exit results, including the replicate run of the unanimity rule (τ=0.67).

Benchmark	Rule	Acc.	ΔAcc.	Speedup	Tok. saved	Calls saved
GSM8K	Full debate	0.92	–	–	–	–
	Unanimity (τ=1.0)	0.95	+0.03	2.11×	59.5%	48.3%
	Unanimity (τ=0.67)	0.96	+0.04	2.02×	57.1%	46.7%
	Majority (τ=0.34)	0.94	+0.02	3.10×	78.1%	61.3%
SVAMP	Full debate	0.92	–	–	–	–
	Unanimity (τ=1.0)	0.93	+0.01	2.18×	60.5%	48.7%
	Unanimity (τ=0.67)	0.92	+0.00	2.08×	60.9%	49.0%
	Majority (τ=0.34)	0.92	+0.00	3.23×	78.5%	61.0%
ASDiv	Full debate	0.95	–	–	–	–
	Unanimity (τ=1.0)	0.94	−0.01	2.91×	73.7%	57.7%
	Unanimity (τ=0.67)	0.95	+0.00	3.01×	75.2%	58.7%
	Majority (τ=0.34)	0.96	+0.01	3.71×	82.3%	63.3%
MultiArith	Full debate	1.00	–	–	–	–
	Unanimity (τ=1.0)	0.99	−0.01	4.82×	87.6%	66.0%
	Unanimity (τ=0.67)	1.00	+0.00	4.85×	87.7%	66.0%
	Majority (τ=0.34)	1.00	+0.00	5.06×	88.7%	66.7%
MATH-500	Full debate	0.45	–	–	–	–
	Unanimity (τ=1.0)	0.43	−0.02	1.23×	22.5%	18.7%
	Unanimity (τ=0.67)	0.38	−0.07	1.14×	16.3%	13.7%
	Majority (τ=0.34)	0.40	−0.05	1.69×	53.5%	43.0%

exact paired McNemar test no early-exit accuracy change is significant ($p \geq 0.125$), and among the downscaling arms only Agg→4B on MATH-500 is (p=0.049).

D.3 Code Workflows

All five code workflows run Qwen3.5-9B with greedy decoding on the 427 sanitized MBPP tasks. Every node that generates code from the task receives the task and its first test case, so that the model knows the expected function signature. The single direct call is one such node. The five-role pipeline runs product manager, architect, project manager, engineer, and QA in sequence, and the QA node reviews the engineer's code against all of the task's test cases and returns the final code. This access favors the two pipeline workflows, which nonetheless score below the direct call. The complexity gate is a rule that marks a task as hard if it exceeds 45 words or contains a hard keyword, and it sends the other tasks from the product manager directly to the engineer and QA (3 calls instead of 5). The dense workflow runs five experts (string, list, dictionary, math, and general) that differ only in a system prompt with the role and domain tips and in one in-domain worked example placed before the task in the user prompt. A selector call then reads the five candidates, without the tests, and returns the most likely correct one. The sparse workflow replaces the five experts and the selector with one expert chosen by a keyword rule, which costs no LLM call. A task passes if the extracted code passes all of its tests within an 8 s timeout.

Table 5: Full node-downscaling results. Cost uses the parameter-weighted token proxy ($9.0\times$ large $+$ $4.0\times$ small tokens).

Benchmark	**Arm**	**Acc.**	**ΔAcc.**	**Speedup**	**Cost red.**
GSM8K	All-9B	0.94	–	–	–
	Agg$\rightarrow$4B	0.93	-0.01	$1.05\times$	14.0%
	Agents$\rightarrow$4B	0.95	$+0.01$	$1.42\times$	40.6%
	All-4B	0.89	-0.05	$1.54\times$	55.2%
SVAMP	All-9B	0.92	–	–	–
	Agg$\rightarrow$4B	0.90	-0.02	$1.05\times$	13.6%
	Agents$\rightarrow$4B	0.92	$+0.00$	$1.48\times$	41.7%
	All-4B	0.89	-0.03	$1.57\times$	55.9%
ASDiv	All-9B	0.95	–	–	–
	Agg$\rightarrow$4B	0.95	$+0.00$	$1.03\times$	13.8%
	Agents$\rightarrow$4B	0.93	-0.02	$1.51\times$	43.2%
	All-4B	0.94	-0.01	$1.59\times$	57.3%
MultiArith	All-9B	1.00	–	–	–
	Agg$\rightarrow$4B	1.00	$+0.00$	$1.02\times$	13.9%
	Agents$\rightarrow$4B	1.00	$+0.00$	$1.43\times$	39.9%
	All-4B	1.00	$+0.00$	$1.49\times$	54.0%
MATH-500	All-9B	0.45	–	–	–
	Agg$\rightarrow$4B	0.36	-0.09	$1.09\times$	15.6%
	Agents$\rightarrow$4B	0.40	-0.05	$1.34\times$	40.3%
	All-4B	0.38	-0.07	$1.35\times$	55.7%