Towards Direct Latent-Space Synthesis for Parallel Branches in LLM-Agent Workflows

Shikun Liu [1] **Mufei Li** [1] **Dongqi Fu** [2] **Haoyu Wang** [1] **Yinglong Xia** [2] **Hong Li** [2] **Hong Yan** [2] **Pan Li** [1]

Abstract

Large language models increasingly serve as execution engines for agentic systems, yet they still consume context through a sequential text interface. This creates a mismatch with modern structured agent workflows, in which independent branches explore subtasks, retrieve evidence, or generate candidate solutions before a final synthesis step. Existing systems typically merge these branches by concatenating their textual outputs, which discards the parallel structure and incurs redundant prefill computation. In this work, we introduce Parallel-Synthesis, a plug-and-play framework that enables a synthesizer to directly consume the KV caches produced by parallel worker agents. Parallel-Synthesis combines a cache mapper that calibrates independently generated branch caches with a fine-tuned synthesizer adapter that enables generation from this non-sequential cache interface. We train Parallel-Synthesis using data that exposes the synthesizer to parallel cache contexts, teaches aggregation across cached branches, and distills reasoning behavior from standard text-concatenation-based synthesis. Across nine downstream datasets spanning math, science QA, code generation, GAIA, and multi-agent database diagnosis, Parallel-Synthesis matches or outperforms text-based synthesis on seven datasets and remains close on the other two. It also reduces time-to-first-token by 2.5×–11×, suggesting that direct cache-based synthesis is a promising interface for more native and efficient synthesis over parallel agent branches.

[1]Georgia Institute of Technology [2]Meta. Correspondence to: Shikun Liu <shikun.liu@gatech.edu>, Pan Li <panli@gatech.edu>.

The Workshop on Graph Foundation Models at the 43rd International Conference on Machine Learning (ICML 2026), Seoul, South Korea. Copyright 2026 by the author(s).

1. Introduction

Large language models (LLMs) serve as the core execution engines for agentic systems due to their advanced reasoning and tool-use capabilities (Yao et al., 2022; Wang et al., 2024b; Schick et al., 2023). Despite these advances, LLMs still operate through a sequential context interface: the entire agent state, including message history, tool outputs, retrieved evidence, and intermediate reasoning, is serialized into a single ordered token prefix for next-step execution. While this interface has been highly effective for scaled-up autoregressive training, it provides only an indirect way to represent the rich dependency structure that arises in agentic computation. Moreover, as histories grow, such serialization also imposes artificial dependencies among otherwise unrelated information, lengthening the prefix that needs to be processed and making it harder for the model to focus on the evidence most relevant to the current query (Liu et al., 2024; Shi et al., 2023; Zhang et al., 2024).

This mismatch becomes especially clear when agent workflows contain parallel structure. Many complex tasks are better represented as directed acyclic graphs (DAGs), where independent subtasks, retrieval branches, candidate solutions, or tool calls can be executed separately and merged later. Such parallel-then-synthesize patterns have emerged as an effective orchestration strategy for improving both reliability and efficiency in complex problem solving. Even in single-agent settings, mathematical reasoning and code generation systems often improve reliability by sampling multiple test-time trajectories and then selecting among candidate solutions or aggregating sampled reasoning chains (Chen et al., 2021; Brown et al., 2024; Wang et al., 2023). In multi-agent systems, this structure becomes more explicit: parallel sub-agents can gather evidence from different perspectives or solve independent subtasks before a downstream agent merges their findings (Anthropic, 2025; Chai et al., 2025; Li et al., 2025; OpenAI, 2026; Anthropic, 2026).

However, because LLMs natively consume only linearized contexts, synthesizing multiple independent branches typically requires concatenating their outputs into a single expanded text prompt for the synthesizer. This text-based communication interface discards the original DAG structure of the computation and incurs redundant prefill cost

by re-encoding information that has already been processed during worker decoding. Since such non-sequential agent workflows become increasingly common, it is timely to move beyond treating context as a single sequential prefix and to consider interfaces that preserve independence among branches while supporting efficient and accurate synthesis.

One natural engineering mitigation is to summarize each branch and pass the summaries to the synthesizer (Anthropic, 2026; Sun et al., 2025; Kang et al., 2025). While this can reduce the long-context burden, it still relies on natural-language communication: summaries must be generated, serialized, and re-encoded through prefill, and the summarization step may discard details that are important for downstream judgment. In this work, we study a more direct alternative, one that points toward a more fundamental agent capability: can a synthesizer agent directly consume the latent states of parallel worker branches, rather than relying on concatenated plain-text outputs? Such an interface could reduce redundant computation while better preserving the parallel structure of non-sequential agent workflows.

Communicating through KV caches instead of explicit text is related to recent work on latent-space communication between agents. Most existing formulations focus on one-to-one latent transfer, where a latent state or cache produced under one context is passed to another agent (Zou et al., 2025; Ye et al., 2025; Fu et al., 2025). Our setting instead studies many-to-one synthesis over parallel agent branches, where the downstream synthesizer must jointly consume and integrate several independently produced branch states. Another adjacent line of work studies parallel encoding and cache reuse for long-context RAG, where retrieved document chunks are independently encoded into KV caches and later reused during generation (Yang et al., 2025b;d; Ratner et al., 2023; Yao et al., 2025). While related in mechanism, this setting differs from ours in how cached contexts are constructed, how KV states are obtained, and what the downstream model is required to do. We defer a more detailed discussion to Sec. 2.2.

In this work, we present **Parallel-Synthesis**, a plug-and-play synthesizer framework for direct synthesis over the cached states of parallel agent branches. Unlike standard text-based synthesis, which concatenates branch outputs and prefills their raw text again, Parallel-Synthesis enables the downstream synthesizer to reuse caches produced by parallel worker agents for subsequent generation. Since this departs from the standard serialization assumed by standard LLM pipelines, Parallel-Synthesis employs a cache mapper to calibrate independently produced branch caches, together with a synthesizer LoRA that adapts the model to this non-sequential cache interface. To support this adaptation, we construct training datasets with complementary goals: 1) large-scale continued-pretraining-style dialogue data for broad adaptation to parallel cache contexts; 2) multi-source synthesis tasks that teach the model to aggregate information from multiple cached contexts when responding to queries; and 3) distillation data from a standard text-concatenation-based synthesis pipeline to strengthen reasoning and judgment over complex agent trajectories.

We evaluate Parallel-Synthesis on downstream tasks spanning standard problem-solving and agentic branch-and-synthesize settings, including mathematical reasoning, scientific QA, code generation, tool-use agentic QA, and multi-agent diagnostic reasoning. Across these diverse settings, Parallel-Synthesis largely preserves or improves the quality of standard text-based synthesis while substantially improving efficiency: it matches or outperforms text-based synthesis on 7 of 9 datasets, with only small gaps on the remaining two, and reduces time-to-first-token (TTFT) by $2.5\times$–$11\times$ by avoiding the re-prefill of branched worker outputs. Notably, Parallel-Synthesis achieves gains on several reasoning-heavy tasks, suggesting that direct latent-state synthesis can be not only faster but also competitive, and sometimes beneficial, for integrating parallel reasoning trajectories.

2. Preliminaries and Problem Formulation

As discussed in Sec. 1 and illustrated in Fig. 1, modern agentic workflows often contain parallel branches executed by independent sub-agents, whose intermediate results must later be merged by a downstream synthesizer. The branching stage is naturally supported by autoregressive decoding: multiple sub-agents can decode from a shared contiguous prefix, forming a tree-structured context. The challenge arises at synthesis, where the downstream agent must condition jointly on several branches rather than continue from a single linear prefix. This merge operation turns the context topology from a tree into a DAG, exposing a mismatch with the standard single-prefix interface of LLMs. This paper studies the resulting problem: how to enable a downstream agent to synthesize information from multiple parallel branches while preserving the DAG-structured context.

2.1. Problem Formulation: Parallel Agent Branching Synthesis

We consider a general parallel-agent workflow with m worker agents followed by a synthesizer. All agents are instantiated from the same autoregressive backbone model f_θ. Each worker agent $j \in \{1, \ldots, m\}$ is conditioned on a branch context c_j and generates an output segment $z_j \sim P_\theta(\cdot \mid c_j)$. We decompose the worker context as $c_j = c^{\mathrm{sh}} \circ c_j^{\mathrm{br}}$, where c^{sh} denotes the context shared across workers, c_j^{br} denotes the branch-specific context, and either may be empty depending on the workflow. After all workers complete, a synthesizer receives its own prompt or instruc-

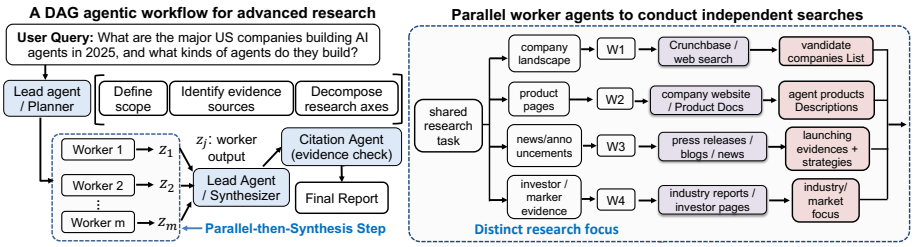

Figure 1. Example of DAG agentic workflow for research containing parallel worker agents searching in different directions.

tion u and generates a final output $\mathbf{y}$ by conditioning jointly on the information produced by the parallel branches.

The central question is how the branch outputs $\{z_j\}_{j=1}^m$ should be communicated to the synthesizer. We compare two synthesis interfaces: the standard text-serialization interface and a cache-based interface that directly reuses the KV states produced by worker agents.

Text serialization. The standard route communicates worker outputs as explicit text. Under a fixed worker ordering, the synthesizer forms a single sequential input prefix $x^{\text{text}} = u \circ z_1 \circ \cdots \circ z_m$ and generates autoregressively as $y_t \sim P_{\text{text}}(\cdot \mid x^{\text{text}} \circ y_{<t})$. This interface is simple and directly compatible with existing LLMs. However, it flattens branch-separated outputs into a single sequence and requires the synthesizer to prefill the worker outputs again, even though these outputs have already been generated by the workers.

Cache-based synthesis. Instead of communicating worker outputs as text, we study whether the synthesizer can directly consume the KV states produced during worker decoding. For worker j, let $\text{KV}_\theta(z_j \mid c_j) = \{(K_{z,j}^\ell, V_{z,j}^\ell)\}_{\ell=1}^L$ denote the layer-wise KV states associated with the output segment z_j generated under context c_j. The synthesizer only prefills its own prompt, $\text{KV}_\theta(u) = \{(K_u^\ell, V_u^\ell)\}_{\ell=1}^L$, and reuses the worker KV states without regenerating them. It then decodes using the directly-concatenated caches $K_{\text{syn}}^\ell = [K_{z,1}^\ell; \cdots; K_{z,m}^\ell; K_u^\ell]$ and $V_{\text{syn}}^\ell = [V_{z,1}^\ell; \cdots; V_{z,m}^\ell; V_u^\ell]$, generating tokens as $y_t \sim P_{\text{kv}}(\cdot \mid \{(K_{\text{syn}}^\ell, V_{\text{syn}}^\ell)\}_{\ell=1}^L, y_{<t})$.

Target. The text-based and cache-based routes expose the same branch outputs $\{z_j\}_{j=1}^m$ to the synthesizer, but through different representations. Our goal is to make cache-based synthesis behave like the standard text-serialization route while avoiding text concatenation and redundant prefill. In particular, for likely generated output sequences $\mathbf{y}$ under the text route, we aim for

$$P_{\text{kv}}\big(\mathbf{y} \mid u, \{\text{KV}_\theta(z_j \mid c_j)\}_{j=1}^m\big) \approx P_{\text{text}}(\mathbf{y} \mid u, z_1, \ldots, z_m)\,.$$

Why this is nontrivial. The cache-based route above goes beyond standard KV-cache reuse. Unlike ordinary autoregressive decoding, where cached states arise from continuing a single prefix, the synthesizer receives caches extracted from disjoint parallel workers, each generated under its own local context. These caches do not fit the ordinary continuation-based KV cache reuse setting for reuse at synthesis time fundamentally nontrivial.

Covered workflow scenarios. The parallel synthesis problem arises across several common agentic workflows. The main difference among these settings lies in how each worker context $c_j = c^{\text{sh}} \circ c_j^{\text{br}}$ is formed before the synthesizer aggregates the branch outputs.

1). Single-turn parallel problem solving. All workers receive the same one-shot task input, such as a math or code-generation problem, and independently generate candidate solutions. In this case, $c_j = c^{\text{sh}}$ for all j, and workers differ only through their independently sampled outputs z_j.

2). Multi-turn parallel trajectory rollout. Workers share the same initial prompt or user query, but each worker accumulates its own reasoning, tool calls, observations, or intermediate messages over time. Here, c^{sh} contains the shared task context, while c_j^{br} contains the worker-specific trajectory. The output z_j represents the final response or summary produced from that trajectory.

3). Distinct sub-task execution. The original task is decomposed into different sub-tasks, each assigned to a separate worker. In this case, c^{sh} may contain the global task description, while c_j^{br} contains the sub-task instruction and any branch-specific trajectory. The output z_j represents the result of the corresponding sub-task branch.

2.2. Difference from RAG-Style Parallel Encoding

Several prior works study KV-cache reuse for long-context RAG, where retrieved documents are independently encoded into cache blocks and later reused for query-conditioned generation (Ratner et al., 2023). To align with our notation, let each retrieved document chunk be denoted by z_j. These methods encode each z_j into a cache block and later reuse the collection of cached chunks during downstream decoding. Since the cached chunks are not produced as a single contiguous prefix, such methods often introduce auxiliary mechanisms, such as partial recomputation (Yao et al., 2025), attention calibration (Yang et al., 2025d), or linking tokens (Yang et al., 2025b), to make independently encoded caches jointly usable.

Our setting is related but differs in several important ways. First, in RAG-style cache reuse, z_j typically denotes an

isolated document chunk encoded without a worker-specific prefix context c_j or prior workflow state. In parallel agent synthesis, by contrast, z_j denotes the output of a worker branch generated under its own context c_j, which may include a local instruction, reasoning trajectory, tool interaction, or intermediate observation. Second, RAG chunks z_j are often obtained by segmenting a longer document (Ratner et al., 2023; Ma et al., 2025), and therefore may not be semantically self-contained. This can leave unresolved dependencies across chunks, requiring mechanisms that repair or calibrate attention across cached blocks. In our setting, z_j is usually a more coherent unit of agent progress, such as a candidate solution, branch summary, or subtask result. This makes branch caches more naturally suited to parallel processing and downstream synthesis than arbitrary document chunks.

The downstream objective is also different. RAG-style cache reuse mainly aims to help the model retrieve and use evidence from cached chunks $\{z_j\}_{j=1}^m$ for a final query. Parallel agent synthesis instead requires the synthesizer to compare, judge, reason over, aggregate, and summarize heterogeneous worker outputs $\{z_j\}_{j=1}^m$ under a broader range of instructions. Thus, while RAG-style cache reuse provides useful technical inspiration, it targets a different communication pattern and downstream objective from the parallel agent branching synthesis setting studied in this work. Consistent with this distinction, our experiments in Sec. 4 show that directly adapting RAG-style cache-reuse methods to parallel agent synthesis leads to weak downstream performance.

3. Methodology

3.1. Model Architecture for the Parallel Synthesizer

Our framework equips the synthesizer with three components: positional re-encoding to realign worker outputs in different branches, a cache mapper to further calibrate these reused caches before synthesis, and a synthesizer-side LoRA adaptation to help the model interpret and reason over multiple branches jointly. The overall pipeline is illustrated in Fig. 2.

Positional re-encoding. Each worker output z_j is originally generated after the full worker-side prefix context c_j. Therefore, if we index the tokens in z_j by their local offset $r = 0, \ldots, |z_j| - 1$, then the r-th token of z_j is cached at RoPE position $|c_j| + r$. Suppose the agent workflow shares a common pre-branch context before spawning parallel worker agents. Let n denote the first RoPE position immediately following this shared prefix at the branching point, the exact step where the original workflow paused. During synthesis, we anchor the first token of every parallel worker's output to position n, ensuring each trajectory structurally branches from the same continuation point rather than being serialized. Concretely, for the r-th token of branch output z_j, we reassign it to the shared target position $n + r$ as for $r = 0, \ldots, |z_j| - 1$:

$$\tilde{k}^{\ell}_{z,j,r} = R(n+r)\, R(|c_j| + r)^{-1} k^{\ell}_{z,j,r},\ \tilde{v}^{\ell}_{z,j,r} = v^{\ell}_{z,j,r}$$

where $R(\cdot)$ denotes the RoPE rotation operator (Su et al., 2024) and obtain re-encoded branch cache $(\tilde{K}^{\ell}_{z,j}, \tilde{V}^{\ell}_{z,j})$.

This positional re-encoding preserves the original token order within each worker agent's output while aligning all worker outputs to the same post-branch positional range. It therefore presents the worker outputs to the synthesizer as parallel continuations from a shared branching point, rather than as an artificially serialized sequence, which may help reduce order-induced positional bias (Wang et al., 2025a;b).

Let $M_{\text{par}} = \max_{j \in [m]} |z_j|$ be the maximum worker output length. After PE re-encoding, all worker outputs lie in the shared interval $[n,\ n{+}M_{\text{par}}{-}1]$, and the synthesizer prompt u is placed after this block. Decoding resumes from position $n + M_{\text{par}} + |u|$, attending to $(K^{\ell}_{\text{syn}}, V^{\ell}_{\text{syn}})$ at every layer.

Cache mapping. After positional re-encoding, we further calibrate the reused worker caches with a learnable cache mapper before they are consumed by the synthesizer. This is related in spirit to APE (Yang et al., 2025d), which uses inference-time attention calibration to make independently encoded contexts better approximate sequential encoding. In contrast, we learn an explicit mapping over cached keys and values jointly with the synthesizer-side LoRA.

Concretely, we form worker-specific metadata $s_j = (|z_j|, m)$, where $|z_j|$ is each worker output length and m is the total number of parallel workers. We learn an MLP that takes the worker statistics s_j as input and predicts, for each layer ℓ, the affine coefficients $\alpha^{\ell}_{K,j}, \beta^{\ell}_{K,j}, \alpha^{\ell}_{V,j}$, and $\beta^{\ell}_{V,j}$. Thus, the resulting transformations remain worker-specific through s_j and layer-specific through ℓ. We then apply them elementwise to the position-re-encoded keys and values before synthesis, which adds negligible overhead compared with re-prefilling worker outputs as text:

$$\hat{K}^{\ell}_{z,j} = \alpha^{\ell}_{K,j}(s_j) \odot \tilde{K}^{\ell}_{z,j} + \beta^{\ell}_{K,j}(s_j),$$
$$\hat{V}^{\ell}_{z,j} = \alpha^{\ell}_{V,j}(s_j) \odot \tilde{V}^{\ell}_{z,j} + \beta^{\ell}_{V,j}(s_j).$$

Ultimately, the synthesizer forms a unified past cache by concatenating the calibrated worker caches with the prefilled cache of its prompt: $K^{\ell}_{\text{syn}} = [\hat{K}^{\ell}_{z,1}; \cdots; \hat{K}^{\ell}_{z,m}; K^{\ell}_{u}]$, $V^{\ell}_{\text{syn}} = [\hat{V}^{\ell}_{z,1}; \cdots; \hat{V}^{\ell}_{z,m}; V^{\ell}_{u}]$.

Synthesizer LoRA adaptation. Cache calibration alone is insufficient. We train the synthesizer via LoRA jointly with the cache mapper. In practice, this forms a plug-and-play synthesizer-side adapter: it is activated only when parallel synthesis is used, while worker-side execution remains unchanged.

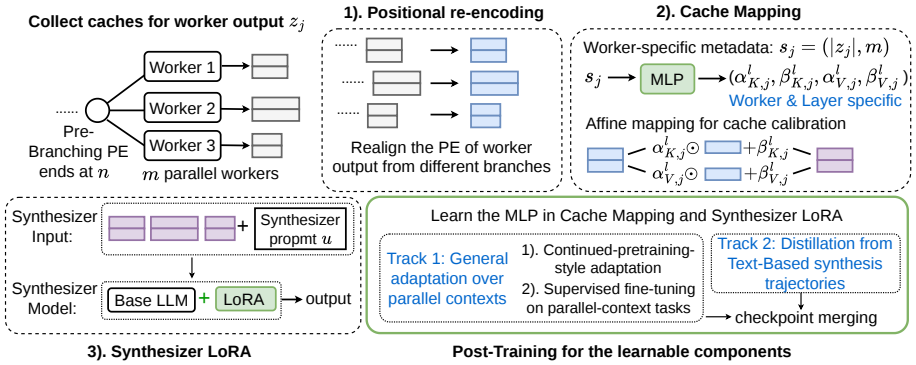

Figure 2. The entire model pipeline with post-training illustration for learnable cache mapping and synthesizer LoRA.

3.2. Model Training

A parallel cache-based synthesizer must acquire two complementary abilities. First, it should learn to *read* and *interpret* parallel worker caches. Second, it should preserve its *reasoning* ability when using those cached trajectories under the synthesizer instruction. We therefore train two complementary post-training tracks from the same base model and merge them afterward: Track 1 provides broad and scaled-up adaptation to parallel cache contexts, while Track 2 targets realistic synthesis and reasoning over parallel agent trajectories through distillation from the text-serialization route.

Training data format and objective. We denote the trainable synthesizer-side parameters by ϕ, including the cache mapper and synthesizer LoRA, while keeping the backbone parameters θ frozen. Following the notation in Sec. 2.1, each training instance is constructed as $(u, \{(c_j, z_j)\}_{j=1}^{m}, y^\star)$, where u is the synthesizer prompt, $\{z_j\}_{j=1}^{m}$ are the parallel worker outputs generated under their corresponding context prefixes $\{c_j\}_{j=1}^{m}$, and $y^\star$ is the supervision target. All training tracks optimize the same teacher-forced next-token objective under the cache-based synthesis route, differing only in how the training instances are constructed: $\mathcal{L}_{\text{SFT}} = -\sum_{t=1}^{|y^\star|} \log P_{\theta,\phi}\left(y^\star_t \mid u, \{\widehat{\text{KV}}_{\theta,\phi}(z_j \mid c_j)\}_{j=1}^{m}, y^\star_{<t}\right)$.

Track 1: General adaptation over parallel contexts. This training track equips the synthesizer to process independently encoded parallel contexts rather than a traditional serialized sequence. We use two complementary data types: continued-pretraining-style data and supervised fine-tuning data on explicit parallel-context tasks. The former teaches the model naturally resume generation from multi-turn histories structured as parallel KV cache. The latter provides explicit, task-oriented supervision, compelling the model to actively synthesize information across these parallel branches to resolve queries. The dataset statistics are included in Appendix A.1.1.

1) Continued-pretraining-style adaptation. We use large-scale multi-turn dialogue data, WildChat (Zhao et al., 2024b), UltraChat (Ding et al., 2023), and LMSYS-Chat (Zheng et al., 2023), to adapt the model to generate from non-sequential prefix caches. We treat the final user query as u and the final assistant response as $y^\star$. The preceding dialogue turns are encoded independently as parallel caches rather than concatenated into one serialized history. Since the target response may depend only weakly on earlier turns, this stage primarily acts as an architectural warm-up that adapts the model to conditioning on parallel contexts.

2) Supervised fine-tuning on parallel-context tasks. We construct training samples from tasks whose inputs can be naturally decomposed into independent, self-contained semantic blocks, and whose solutions explicitly require the model to follow the instruction and read and understand the provided parallel contexts. For agentic tool-use data such as Toucan (Xu et al., 2025) and DTA-Tool (Zhu et al., 2025a), we extract parallel tool-use trajectories. Here, u is the user query, each tool-call–observation pair is treated as a parallel branch output z_j, the preceding agent history defines its prefix context c_j, and $y^\star$ is the next agent response that interprets and uses these tool results. For in-context learning data like FLAN (Wei et al., 2022), u is the target question, $y^\star$ is the answer, and preceding few-shot exemplars serve as the parallel contexts. Finally, for multi-document QA such as 2Wiki (Ho et al., 2020), u is the query and individual source documents form the parallel branches. We construct $y^\star$ as a reasoning trace that first summarizes the provided contexts, analyzes their relevance to u, and then deduces the target answer.

Track 2: Distillation from Text-Based Synthesis Trajectories. This track targets the richer synthesis behavior required in realistic agentic settings, where the synthesizer must reason over, compare, and judge prior sub-agent progress under detailed task instructions. Although Track 1 teaches the model to read and use parallel caches, it does not provide enough supervision for higher-level reasoning over the evidence encoded in them, which limits end-task synthesis performance in practice. We therefore distill synthesis trajectories from the text-serialization route, using them as teacher signals for reasoning-intensive synthesis.

Concretely, we use BrowseComp (Wei et al., 2025) to con-

struct parallel worker trajectories and synthesis supervision. For each sample, we first generate three parallel worker rollouts and then run the synthesizer under the text-based synthesis route, where the worker outputs are serialized as text context. Although these synthesis traces are constructed from three worker trajectories, the trained synthesizer is not tied to this rollout count and generalizes to different numbers of parallel trajectories at evaluation time. We then collect the resulting synthesis trajectories and filter out failed or low-quality cases, retaining only those with reliable reasoning and valid final responses as supervision targets $y^\star$ for cache-based training. Each retained example is converted into our cache-based training format: for worker j, c_j is the worker-local history before its final output turn, including prior messages, tool calls, tool observations, and intermediate reasoning, and z_j is the final output communicated to the synthesizer. The synthesizer prompt u is kept unchanged from the text-based synthesis route.

Checkpoint merging. We train the two tracks independently and merge them afterward, since sequential tuning gives weaker results (Sec. 4.2), suggesting partial overwriting of earlier capabilities. This matches findings that continued post-training can induce forgetting, whereas merging separately specialized checkpoints better preserve complementary updates (Yang et al., 2025c; Xiao et al., 2024; Harmon et al., 2025; He et al., 2025). We merge the parameters of the cache mapper and the LoRA by weighted averaging, with $\lambda = 0.5$ by default.

4. Experiments

Evaluation datasets We evaluate Parallel-Synthesis on nine datasets across four domains: (1) Mathematical reasoning: AIME 2024/2025 (Maxwell-Jia, 2024; Zhang & Math-AI, 2025) and GSM8K (Cobbe et al., 2021); (2) Scientific QA: GPQA (Rein et al., 2024) and MedQA (Yang et al., 2025a); (3) Code generation: HumanEval-Plus and MBPP-Plus (Liu et al., 2023); and (4) Multi-turn agentic benchmarks with tool use: GAIA (Mialon et al., 2023) and the database task from MARBLE (Zhu et al., 2025b). In all tasks except MARBLE Database, parallel worker agents are used to explore multiple candidate solution or search trajectories. In MARBLE Database, worker agents instead handle decomposed subtasks to independently analyze possible root causes of database errors. We use 5 parallel worker agents for MARBLE Database task since it is determined by fixed sub-tasks and 3 parallel workers for the rest datasets. These benchmarks cover diverse settings where parallel-then-synthesize execution is useful. Importantly, none of them overlap with the post-training datasets, so all results reflect out-of-domain generalization. More details are provided in Appendix A.1.2.

Models and Baselines. We use Qwen3-14B as the backbone for Parallel-Synthesis and, whenever possible, for all

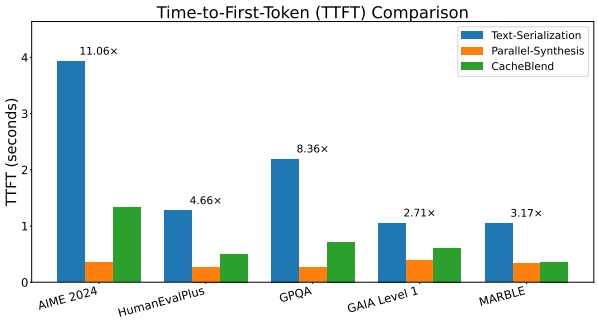

Figure 3. TTFT comparison between Text-Serialization, CacheBlend and Parallel-Synthesis. The label indicate speedup to Text-Serialization.

baselines. We organize baselines into three groups. *Single-trajectory and answer-level aggregation* baselines test whether gains come simply from sampling multiple workers: *Single* uses one agent trajectory and its final answer, while *Voting* aggregates only the final answers $\hat{y}_j$ extracted from worker outputs z_j via $\hat{y} = \arg\max_a \sum_{j=1}^{m} \mathbf{1}[\hat{y}_j = a]$, counting ties as incorrect. We omit these two baselines for MARBLE Database, which is inherently a decomposed multi-agent benchmark. *Text-based synthesis* is represented by *Text-Serialization* from Sec. 2.1, which uses the same worker outputs as Parallel-Synthesis but communicates them by concatenating their textual outputs into the synthesizer prompt. *Cache-based synthesis* baselines adapt RAG-style parallel-encoding methods, including APE (Yang et al., 2025d), CacheBlend (Yao et al., 2025), and KVLINK (Yang et al., 2025b): APE calibrates attention at inference time, CacheBlend selectively recomputes KV states, and KVLINK uses linking tokens between cache blocks with a post-trained synthesizer. For KVLINK, we use the largest released checkpoint; worker trajectories are generated by Qwen3-14B and then encoded into KVLINK's linked-cache format for final synthesis. Additional evaluation details are provided in Appendix A.2.

4.1. Main Results

RQ1: How does Parallel-Synthesis compare to Text-Serialization in performance and efficiency?

Based on Table 1, Parallel-Synthesis matches or surpasses Text-Serialization on 7 of 9 datasets, with only small gaps on the remaining two. This shows that, after post-training, the synthesizer can reason effectively over prior agents' outputs without requiring those outputs to be reintroduced as text in the prompt. The gains are pronounced on reasoning-heavy datasets, including math and science benchmarks and the MARBLE database task, where the synthesizer must integrate rich reasoning traces from parallel workers. This suggests that explicit text can be a limited communication interface for reasoning-heavy content, whereas cache sharing may preserve richer latent reasoning states for synthesis. This aligns with prior work on latent-space reasoning (Hao et al., 2025; Zheng et al., 2025; Su et al., 2025). By contrast,

Table 1. Performance across math, code generation, scientific QA, and agentic benchmarks. Math, code generation, and scientific QA results are reported in Accuracy. For GAIA and MARBLE, each entry reports the number of correct samples. The total numbers of samples are 53, 86, and 26 for GAIA Levels 1–3, respectively, and 100 for MARBLE Database. We **bold** where Parallel-Synthesis matches or outperforms Text-Serialization.

Method	Math problem			Code generation		Scientific QA		GAIA			MARBLE
	AIME 2024	AIME 2025	GSM8K	HumanEvalPlus	MBPPPlus	GPQA	MedQA	Level 1	Level 2	Level 3	Database
Single	23.33	13.33	92.42	84.15	76.19	31.31	82.72	14	18	1	N/A
Voting	30.00	20.00	92.20	89.63	79.37	48.99	83.74	19	16	1	N/A
CacheBlend	13.33	23.33	79.15	51.22	52.91	12.63	40.38	17	16	0	7
APE	6.67	10.00	49.66	36.59	52.91	10.10	17.05	20	13	2	18
KVLINK	50.00	40.00	92.72	87.80	78.31	35.35	67.05	20	20	1	10
Text-Serialization	63.33	23.33	92.80	90.24	81.75	50.00	82.72	24	22	2	33
Parallel-Synthesis	**63.33**	**46.67**	**94.69**	**90.85**	80.42	**52.02**	**83.58**	23	19	**2**	**36**

explicit text can still be advantageous in more evidence-centric cases, such as GAIA examples involving search results or MBPP-Plus problems with explicit code outputs, where text may provide a clearer and more robust evidence representation.

To quantify the computational savings of Parallel-Synthesis from avoiding the re-prefill of parallel worker outputs required by Text-Serialization, we measure the synthesizer's Time-to-First-Token (TTFT) on five datasets. For fairness, we include in Parallel-Synthesis 's latency both the cache transformation time, covering cache mapping, positional re-encoding, and the TTFT. Fig. 3 shows that Parallel-Synthesis consistently reduces latency across all datasets, achieving 2.5×–11× speedups over Text-Serialization. We also compare with CacheBlend using a 0.15 selective-recomputation ratio, Parallel-Synthesis remains an average 2× faster while achieving higher downstream performance. These gains become larger as worker agent output trajectories grow longer.

RQ2: Does Parallel-Synthesis perform genuine reasoning-based synthesis over parallel agent trajectories, beyond simple answer aggregation?

An important question is whether Parallel-Synthesis uses parallel agent trajectories as rich synthesis context, rather than treating them as a set of final answers to be aggregated by voting. The *Voting* baseline provides a direct test for this distinction. Across 8 of the 9 datasets, Parallel-Synthesis outperforms pure majority voting, indicating that its gains cannot be explained by simply recovering the most common worker answer. MedQA is the only exception, as this multiple-choice task is amenable to majority voting.

These results suggest that Parallel-Synthesis can exploit trajectory-level signals beyond workers' final answers, including intermediate reasoning steps, evidence quality, and disagreements among workers. This capability is especially important for tasks such as GAIA, where the correct answer is not always supported by the majority of workers' final answers. Case studies in Appendix A.4 show that Parallel-Synthesis can ground its synthesis in intermediate evidence from parallel agents; in some examples, it even performs an additional tool call for verification when needed. By contrast, although KVLINK performs reasonably on some datasets, it lags behind Parallel-Synthesis overall and often jumps directly to a final answer without reasoning over the worker trajectories or provided instruction. This suggests that post-training for RAG-style cache reuse may not be sufficient for the trajectory-level reasoning and instruction-following ability needed for parallel-agent synthesis.

Table 2. Ablation study on checkpoints and model modules. The full default model uses the merged Training Track 1+2 checkpoint together with cache mapping and Synthesizer LoRA.

Ablation	Variant	AIME 2024	HumanEvalPlus	GPQA	GAIA Level 1	MARBLE
Checkpoint	No training	10.00	38.41	15.15	16/53	22/100
	Track 1	46.67	91.46	55.56	21/53	30/100
	Track 2	53.33	88.41	51.52	21/53	44/100
	Sequential Track 1+2	36.67	76.22	49.49	23/53	25/100
Module	Synthesizer LoRA only	60.00	88.41	40.40	22/53	35/100
Full merged model checkpoint (default)		63.33	90.85	52.02	23/53	36/100

Table 3. Ablation study on the information passed to the synthesizer on GAIA Level 1. Correctness reports the number of correct samples out of 53 samples. TTFT is measured in seconds.

Information for synthesizer	Correctness		TTFT (s)	
	Parallel-Synthesis	Text-Serialization	Parallel-Synthesis	Text-Serialization
Final model output	23	24	0.3769	1.0510
Each-turn model output	18	22	0.3788	1.5053
Full trajectory	26	26	0.6847	3.5383

RQ3: Importance of post-training for robust and general parallel cache synthesis.

Post-training is critical for making direct cache reuse reliable in parallel synthesis. Although Parallel-Synthesis is not trained on any of the nine downstream benchmarks, it transfers effectively across all of them, suggesting that the learned synthesis behavior generalizes beyond the training distribution. In contrast, inference-time cache-reuse methods such as APE, CacheBlend, and the no-training variant of Parallel-Synthesis perform substantially worse, whereas the post-trained KVLINK baseline is comparatively stronger. This shows that robust parallel cache synthesis requires explicit training-based adaptation, rather than direct inference-time cache calibration alone. Sec. 4.2 further analyzes the contribution of training data and design choices.

4.2. Ablation Studies

Ablation on different post-training checkpoints. We compare four variants on one representative dataset from each task category: no post-training, Track 1 only, Track 2 only, and sequential tuning where Track 2 is applied after Track 1. As shown in Table 2, Track 1 tends to perform

better on simpler single-turn problem-solving tasks, such as HumanEvalPlus and GPQA, whereas Track 2 is stronger on reasoning-heavy math and more complex agentic tasks. This pattern is consistent with the data composition and intended capabilities of the two training tracks described in Sec. 3.2. The final merged checkpoint achieves the best overall balance, while sequential tuning typically underperforms both single-track checkpoints and the merged model. This suggests that sequential tuning fails to preserve the complementary strengths learned from the two tracks.

Ablation on framework modules. We further examine the contributions of the learnable cache mapper and synthesizer LoRA. Since the no-training variant and inference-time cache-reuse baseline APE perform poorly, synthesizer adaptation is necessary for parallel cache synthesis. Table 2 further shows that adding the learnable cache mapper consistently improves over the synthesizer-LoRA-only variant, indicating that cache mapping provides complementary benefits beyond LoRA adaptation.

Ablation on communicated trajectory information. In single-turn tasks, each worker produces a single response, so the communicated context is unambiguous. In multi-turn ReAct-style tasks, however, a worker trajectory contains instructions, intermediate model responses, tool calls, and tool observations, raising the question of which parts should be passed to the synthesizer. Table 3 compares three choices on GAIA Level 1: *final output*, which passes only the worker's final response; *each-turn output*, which passes all model-generated responses but excludes instructions and tool observations; and *full trajectory*, which passes the complete worker history. Full trajectories achieve the highest accuracy by preserving tool observations, but incur the largest latency, especially for Text-Serialization. Final outputs provide the best accuracy-efficiency tradeoff and are used as our default. Each-turn outputs perform worse, likely because removing tool observations leaves fragmented, weakly grounded traces. These results show that Parallel-Synthesis can synthesize from different levels of trajectory information while benefiting most from concise, coherent worker outputs.

5. More Related Works

LLM-based Multi-agent Systems (MAS). Recent LLM-based multi-agent systems move beyond sequential execution (Yao et al., 2022; Zhang et al., 2024) by organizing agent computation as structured workflows, often in the form of computational graphs or DAGs, where LLM/tool nodes correspond to intermediate subtasks and edges specify execution dependencies (Zhuge et al., 2024; Zhang et al., 2025b; Yu et al., 2025; Qin et al., 2025). Within these graph-structured workflows, the parallel-then-synthesize pattern appears in several forms. First, some workflows parallelize solution and rationale generation, where multiple agents or model calls independently solve the same problem and their candidate answers are later aggregated through voting, debate, ranking, or response fusion (Du et al., 2024; Li et al., 2024; Jiang et al., 2023; Wang et al., 2024a). Second, some workflows parallelize tool use and open-ended exploration, where independent retrieval/API calls or tool-using subagents are executed along different search directions before a lead agent synthesizes their observations (Kim et al., 2024; Anthropic, 2025; Wu et al., 2025; Miloradović et al., 2023; Zhang et al., 2025c; Chai et al., 2025). Lastly, some works parallelize over input contexts: long documents or retrieved evidence are split into chunks, processed by separate agents and aggregated into a final answer (Zhao et al., 2024a; Zhou et al., 2025; Liu et al., 2025).

Latent communication in multi-agent systems. Many work studies how LLM agents can communicate beyond decoded texts. They mainly fall into two lines: one uses latent representations as compact thought-level messages for direct agent communication (Zheng et al., 2025; Zou et al., 2025; Du et al., 2025), while the other treats KV caches as transferable execution states, enabling agents or models to reuse, share, or fuse internal context without decoding it into text (Ye et al., 2025; Shi et al., 2025; Fu et al., 2025). These works mainly study pairwise or sequential inter-agent transfer, where one agent's internal state is passed to another agent. In contrast, our setting targets the many-to-one convergence pattern in non-sequential workflows: several independently generated branch caches must be jointly interpreted by a downstream synthesizer. A recent work (Jin et al., 2026) investigates reusable latent primitives for multi-agent collaboration, including parallel branches voting and selection, whereas we focus on training a general parallel-agent synthesizer.

6. Conclusion and Future Direction

We introduced Parallel-Synthesis, a plug-and-play framework that enables a downstream synthesizer to directly consume the KV caches produced by parallel worker agents. Through post-training, Parallel-Synthesis performs effectively across a diverse set of parallel-then-synthesize tasks, matching or surpassing text-based synthesis while substantially reducing redundant computation. Parallel-Synthesis is not yet ready to replace text-based communication in all agent frameworks, especially for highly complex structured workflows. Nevertheless, our results provide encouraging evidence that direct cache-based synthesis is a promising interface for parallel agent branches. If this direction succeeds, DAG-structured workflows could be executed and merged more natively, without the repeated overhead of text serialization and context refilling. Achieving this goal will likely require scaling post-training to broader and larger collections of agent workflow data, covering more diverse tool-use patterns, branching structures, and synthesis objectives.

Acknowledgments

S. Liu, M. Li, H. Wang, and P. Li are partially supported by the NSF under awards IIS-2239565, CCF-2402816, IIS-2435957; the Meta Grant; the NVIDIA Academic Grant Program; and the IDEaS Cyberinfrastructure Awards.

References

Anthropic. How we built our multi-agent research system. https://www.anthropic.com/engineering/multi-agent-research-system, June 2025. Anthropic Engineering Blog.

Anthropic. Create custom subagents. https://code.claude.com/docs/en/sub-agents, 2026. Claude Code documentation.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Brown, B., Juravsky, J., Ehrlich, R., Clark, R., Le, Q. V., Ré, C., and Mirhoseini, A. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

Chai, J., Tang, S., Ye, R., Du, Y., Zhu, X., Zhou, M., Wang, Y., Zhang, Y., Zhang, L., Chen, S., et al. Scimaster: Towards general-purpose scientific ai agents, part i. x-master as foundation: Can we lead on humanity's last exam? *arXiv preprint arXiv:2507.05241*, 2025.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Ding, N., Chen, Y., Xu, B., Qin, Y., Zheng, Z., Hu, S., Liu, Z., Sun, M., and Zhou, B. Enhancing chat language models by scaling high-quality instructional conversations, 2023.

Du, Y., Li, S., Torralba, A., Tenenbaum, J. B., and Mordatch, I. Improving factuality and reasoning in language models through multiagent debate. In *Forty-first international conference on machine learning*, 2024.

Du, Z., Wang, R., Bai, H., Cao, Z., Zhu, X., Cheng, Y., Zheng, B., Chen, W., and Ying, H. Enabling agents to communicate entirely in latent space. *arXiv preprint arXiv:2511.09149*, 2025.

Fu, T., Min, Z., Zhang, H., Yan, J., Dai, G., Ouyang, W., and Wang, Y. Cache-to-cache: Direct semantic communication between large language models. *arXiv preprint arXiv:2510.03215*, 2025.

Hao, S., Sukhbaatar, S., Su, D., Li, X., Hu, Z., Weston, J. E., and Tian, Y. Training large language models to reason in a continuous latent space. In *Second Conference on Language Modeling*, 2025.

Harmon, J., Hochlehnert, A., Bethge, M., and Prabhu, A. Mapping post-training forgetting in language models at scale. In *AI That Keeps Up: NeurIPS 2025 Workshop on Continual and Compatible Foundation Model Updates*, 2025.

He, Y., Zeng, S., Hu, Y., Yang, R., Zhang, T., and Zhao, H. Mergebench: A benchmark for merging domain-specialized llms. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2025.

Ho, X., Duong Nguyen, A.-K., Sugawara, S., and Aizawa, A. Constructing a multi-hop QA dataset for comprehensive evaluation of reasoning steps. In *Proceedings of the 28th International Conference on Computational Linguistics*, pp. 6609–6625, Barcelona, Spain (Online), December 2020. International Committee on Computational Linguistics. URL https://www.aclweb.org/anthology/2020.coling-main.580.

Jiang, D., Ren, X., and Lin, B. Y. Llm-blender: Ensembling large language models with pairwise ranking and generative fusion. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14165–14178, 2023.

Jin, H., Peng, K., Yu, Y., Yuan, X., and Wang, H. Agent primitives: Reusable latent building blocks for multi-agent systems. *arXiv preprint arXiv:2602.03695*, 2026.

Kang, M., Chen, W.-N., Han, D., Inan, H. A., Wutschitz, L., Chen, Y., Sim, R., and Rajmohan, S. Acon: Optimizing context compression for long-horizon llm agents. *arXiv preprint arXiv:2510.00615*, 2025.

Kim, S., Moon, S., Tabrizi, R., Lee, N., Mahoney, M. W., Keutzer, K., and Gholami, A. An llm compiler for parallel function calling. In *Forty-first International Conference on Machine Learning*, 2024.

Li, B., Zhang, D., Wu, J., Yin, W., Tao, Z., Zhao, Y., Zhang, L., Shen, H., Fang, R., Xie, P., et al. Parallelmuse: Agentic parallel thinking for deep information seeking. *arXiv preprint arXiv:2510.24698*, 2025.

Li, J., Zhang, Q., Yu, Y., Fu, Q., and Ye, D. More agents is all you need. *arXiv preprint arXiv:2402.05120*, 2024.

Liu, J., Xia, C. S., Wang, Y., and Zhang, L. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572, 2023.

Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., and Liang, P. Lost in the middle: How language models use long contexts. *Transactions of the association for computational linguistics*, 12:157–173, 2024.

Liu, Z., Wan, Z., Li, P., Yan, M., Huang, F., and Liu, Y. Scaling external knowledge input beyond context windows of llms via multi-agent collaboration. *arXiv preprint arXiv:2505.21471*, 2025.

Ma, D., Wang, Y., and Lan, T. Block-attention for efficient prefilling. In *The Thirteenth International Conference on Learning Representations*, 2025.

Maxwell-Jia. Aime 2024 dataset. https://huggingface.co/datasets/Maxwell-Jia/AIME_2024, 2024. Hugging Face dataset.

Mialon, G., Fourrier, C., Wolf, T., LeCun, Y., and Scialom, T. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*, 2023.

Miloradović, B., Çürüklü, B., Ekström, M., and Papadopoulos, A. V. Optimizing parallel task execution for multi-agent mission planning. *IEEE Access*, 11:24367–24381, 2023.

Nguyen, D., Lai, V. D., Yoon, S., Rossi, R. A., Zhao, H., Zhang, R., Mathur, P., Lipka, N., Wang, Y., Bui, T., et al. Dynasaur: Large language agents beyond predefined actions. In *Second Conference on Language Modeling*, 2025.

OpenAI. Subagents. https://developers.openai.com/codex/subagents, 2026. OpenAI Developers, Codex documentation.

Qin, T., Chen, Q., Wang, S., Xing, H., Zhu, K., Zhu, H., Shi, D., Liu, X., Zhang, G., Liu, J., et al. Flash-searcher: Fast and effective web agents via dag-based parallel execution. *arXiv preprint arXiv:2509.25301*, 2025.

Ratner, N., Levine, Y., Belinkov, Y., Ram, O., Magar, I., Abend, O., Karpas, E., Shashua, A., Leyton-Brown, K., and Shoham, Y. Parallel context windows for large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6383–6402, 2023.

Rein, D., Hou, B. L., Stickland, A. C., Petty, J., Pang, R. Y., Dirani, J., Michael, J., and Bowman, S. R. Gpqa: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*, 2024.

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N., and Scialom, T. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 2023.

Shi, F., Chen, X., Misra, K., Scales, N., Dohan, D., Chi, E. H., Schärli, N., and Zhou, D. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning*, pp. 31210–31227. PMLR, 2023.

Shi, X., Chiesa, M., Maguire Jr, G. Q., and Kostic, D. Kvcomm: Enabling efficient llm communication through selective kv sharing. *arXiv preprint arXiv:2510.03346*, 2025.

Soni, A. B., Li, B., Wang, X., Chen, V., and Neubig, G. Coding agents with multimodal browsing are generalist problem solvers. In *Findings of the Association for Computational Linguistics: EACL 2026*, pp. 6052–6069, 2026.

Su, D., Zhu, H., Xu, Y., Jiao, J., Tian, Y., and Zheng, Q. Token assorted: Mixing latent and text tokens for improved language model reasoning. In *International Conference on Machine Learning*, pp. 57144–57163. PMLR, 2025.

Su, J., Ahmed, M., Lu, Y., Pan, S., Bo, W., and Liu, Y. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

Sun, W., Lu, M., Ling, Z., Liu, K., Yao, X., Yang, Y., and Chen, J. Scaling long-horizon llm agent via context-folding. *arXiv preprint arXiv:2510.11967*, 2025.

Wang, H. P., Wang, P., Li, M., Liu, S., Miao, S., Wang, Z., and Li, P. Graph-kv: Breaking sequence via injecting structural biases into large language models. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025a.

Wang, J., Jue, W., Athiwaratkun, B., Zhang, C., and Zou, J. Mixture-of-agents enhances large language model capabilities. In *The Thirteenth International Conference on Learning Representations*, 2024a.

Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Chen, X., Lin, Y., et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024b.

Wang, X., Wei, J., Schuurmans, D., Le, Q. V., Chi, E. H., Narang, S., Chowdhery, A., and Zhou, D. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023.

Wang, Z., Zhang, H., Li, X., Huang, K.-H., Han, C., Ji, S., Kakade, S. M., Peng, H., and Ji, H. Eliminating position bias of language models: A mechanistic approach. In *The Thirteenth International Conference on Learning Representations*, 2025b.

Wei, J., Bosma, M., Zhao, V., Guu, K., Yu, A. W., Lester, B., Du, N., Dai, A. M., and Le, Q. V. Finetuned language models are zero-shot learners. In *International Conference on Learning Representations*, 2022.

Wei, J., Sun, Z., Papay, S., McKinney, S., Han, J., Fulford, I., Chung, H. W., Passos, A. T., Fedus, W., and Glaese, A. Browsecomp: A simple yet challenging benchmark for browsing agents. *arXiv preprint arXiv:2504.12516*, 2025.

Wu, J., Zhao, Q., Chen, Z., Qin, K., Zhao, Y., Wang, X., and Yao, Y. Gap: Graph-based agent planning with parallel tool use and reinforcement learning. *arXiv preprint arXiv:2510.25320*, 2025.

Xiao, S., Liu, Z., Zhang, P., and Xing, X. Lm-cocktail: Resilient tuning of language models via model merging. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 2474–2488, 2024.

Xu, Z., Soria, A. M., Tan, S., Roy, A., Agrawal, A. S., Poovendran, R., and Panda, R. Toucan: Synthesizing 1.5m tool-agentic data from real-world mcp environments, 2025. URL https://arxiv.org/abs/2510.01179.

Yang, H., Chen, H., Guo, H., Chen, Y., Lin, C.-S., Hu, S., Hu, J., Wu, X., and Wang, X. Llm-medqa: Enhancing medical question answering through case studies in large language models. In *2025 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8. IEEE, 2025a.

Yang, J., Hou, B., Wei, W., Bao, Y., and Chang, S. Kvlink: Accelerating large language models via efficient kv cache reuse. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025b.

Yang, J., Jin, D., Tang, A., Shen, L., Zhu, D., Chen, Z., Zhao, Z., Wang, D., Cui, Q., Zhang, Z., et al. Mix data or merge models? balancing the helpfulness, honesty, and harmlessness of large language model via model merging. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025c.

Yang, X., Chen, T., and Chen, B. Ape: Faster and longer context-augmented generation via adaptive parallel encoding. In *The Thirteenth International Conference on Learning Representations*, 2025d.

Yao, J., Li, H., Liu, Y., Ray, S., Cheng, Y., Zhang, Q., Du, K., Lu, S., and Jiang, J. Cacheblend: Fast large language model serving for rag with cached knowledge fusion. In *Proceedings of the twentieth European conference on computer systems*, pp. 94–109, 2025.

Yao, S., Zhao, J., Yu, D., Shafran, I., Narasimhan, K. R., and Cao, Y. React: Synergizing reasoning and acting in language models. In *NeurIPS 2022 Foundation Models for Decision Making Workshop*, 2022.

Ye, H., Gao, Z., Ma, M., Wang, Q., Fu, Y., Chung, M.-Y., Lin, Y., Liu, Z., Zhang, J., Zhuo, D., et al. Kvcomm: Online cross-context kv-cache communication for efficient llm-based multi-agent systems. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Yu, J., Ding, Y., and Sato, H. Dyntaskmas: A dynamic task graph-driven framework for asynchronous and parallel llm-based multi-agent systems. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 35, pp. 288–296, 2025.

Zhang, E., Zhu, E., Bansal, G., Fourney, A., Mozannar, H., and Gerrits, J. Optimizing sequential multi-step tasks with parallel llm agents. *arXiv preprint arXiv:2507.08944*, 2025a.

Zhang, J., Xiang, J., Yu, Z., Teng, F., Chen, X.-H., Chen, J., Zhuge, M., Cheng, X., Hong, S., Wang, J., et al. Aflow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations*, 2025b.

Zhang, W., Cui, C., Zhao, Y., Hu, R., Liu, Y., Zhou, Y., and An, B. Agentorchestra: A hierarchical multi-agent framework for general-purpose task solving. *arXiv e-prints*, pp. arXiv–2506, 2025c.

Zhang, Y. and Math-AI, T. American invitational mathematics examination (aime) 2025. https://huggingface.co/datasets/math-ai/aime25, 2025. Hugging Face dataset.

Zhang, Y., Sun, R., Chen, Y., Pfister, T., Zhang, R., and Arık, S. Ö. Chain of agents: Large language models collaborating on long-context tasks. *Advances in Neural Information Processing Systems*, 37:132208–132237, 2024.

Zhao, J., Zu, C., Hao, X., Lu, Y., He, W., Ding, Y., Gui, T., Zhang, Q., and Huang, X.-J. Longagent: achieving question answering for 128k-token-long documents through multi-agent collaboration. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 16310–16324, 2024a.

Zhao, W., Ren, X., Hessel, J., Cardie, C., Choi, Y., and Deng, Y. Wildchat: 1m chatGPT interaction logs in the wild. In *The Twelfth International Conference on Learning Representations*, 2024b. URL https://openreview.net/forum?id=Bl8u7ZRlbM.

Zheng, L., Chiang, W.-L., Sheng, Y., Li, T., Zhuang, S., Wu, Z., Zhuang, Y., Li, Z., Lin, Z., Xing, E. P., Gonzalez, J. E., Stoica, I., and Zhang, H. Lmsys-chat-1m: A large-scale real-world llm conversation dataset, 2023.

Zheng, Y., Zhao, Z., Li, Z., Xie, Y., Gao, M., Zhang, L., and Zhang, K. Thought communication in multiagent collaboration. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Zhou, Z., Li, C., Chen, X., Wang, S., Chao, Y., Li, Z., Wang, H., Shi, Q., Tan, Z., Han, X., et al. Llm× mapreduce: Simplified long-sequence processing using large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 27664–27678, 2025.

Zhu, D., Shi, W., Shi, Z., Ren, Z., Wang, S., Yan, L., and Yin, D. Divide-then-aggregate: An efficient tool learning method via parallel tool invocation, 2025a. URL https://arxiv.org/abs/2501.12432.

Zhu, K., Du, H., Hong, Z., Yang, X., Guo, S., Wang, D. Z., Wang, Z., Qian, C., Tang, R., Ji, H., et al. Multiagentbench: Evaluating the collaboration and competition of llm agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8580–8622, 2025b.

Zhuge, M., Wang, W., Kirsch, L., Faccio, F., Khizbullin, D., and Schmidhuber, J. Gptswarm: Language agents as optimizable graphs. In *Forty-first International Conference on Machine Learning*, 2024.

Zou, J., Yang, X., Qiu, R., Li, G., Tieu, K., Lu, P., Shen, K., Tong, H., Choi, Y., He, J., et al. Latent collaboration in multi-agent systems. *arXiv preprint arXiv:2511.20639*, 2025.

A. Technical appendices and supplementary material

We provide additional details on the datasets used for post-training and downstream evaluation in Appendix A.1. Appendix A.2 reports the experimental configurations, and Appendix A.3 specifies the prompts used for each dataset. Finally, Appendix A.4 presents case studies that qualitatively illustrate the reasoning capability of the trained Parallel-Synthesis when synthesizing information from parallel agent trajectories.

A.1. Datasets Details

This section will include omitted detail of the dataset introduction and some statistics of the post-training datasets as well as the downstream evaluation datasets.

A.1.1. Post-training datasets

The statistics for the post-training datasets are included in Table 4.

- Wildchat (Zhao et al., 2024b): A large-scale corpus of real-world user–ChatGPT conversations collected from opt-in users. It contains diverse user intents, languages, and multi-turn interaction patterns, making it useful for studying realistic dialogue histories.
- LMSYS-Chat (Zheng et al., 2023): A real-world LLM conversation dataset collected from the Vicuna demo and Chatbot Arena. It contains conversations with multiple state-of-the-art LLMs and is useful for modeling natural user–assistant interactions.
- UltraChat (Ding et al., 2023): A large-scale synthetic multi-turn instructional dialogue dataset generated with ChatGPT-style user and assistant roles. It covers broad open-domain instruction-following scenarios and is designed for chat model training.
- Toucan-single-turn (Xu et al., 2025): A subset of Toucan-1.5M, a large tool-agentic dataset synthesized from real-world MCP environments. Here, we filter out only single-turn interaction setting but with parallel toolcalls. We train the model to generate corresponding response given the parallel tool-calling execution.
- Toucan-multi-turn (Xu et al., 2025): The multi-turn subset of Toucan, containing tool-agent trajectories extended to multi-turn interactions. It is useful for training models to plan tool use conditioned on previous user–agent interactions.
- DTA-Tool (Zhu et al., 2025a): A parallel tool invocation dataset built for DTA-Llama. It converts traditional tool-search trajectories into DAG-style parallel tool invocation traces, where the model learns to divide a task into parallel tool-using subtasks and aggregate the results.
- FLAN (Wei et al., 2022): An instruction-tuning collection that formats many NLP tasks with natural language instructions, including zero-shot, few-shot, and chain-of-thought templates. In your setting, it is suitable for constructing current-question/current-answer pairs with previous in-context examples.
- 2WikiMultiHopQA (Ho et al., 2020): A multi-hop question answering dataset built from structured Wikidata and unstructured Wikipedia evidence. It provides explicit evidence and reasoning paths, making it useful for training or evaluating context-grounded multi-hop reasoning.

A.1.2. Evaluation datasets

Math Problem Reasoning

- AIME (Maxwell-Jia, 2024; Zhang & Math-AI, 2025): AIME 2024 and 2025 each contains 30 competition-level problems from the American Invitational Mathematics Examination 2024 and 2025. The questions are span algebra, geometry, number theory, and combinatorics, and require precise numeric answers with typically 1–3 digits and are intended for evaluating LLM mathematical reasoning on challenging high-school olympiad-style questions.
- GSM8K (Cobbe et al., 2021): It consists of 8.5K human-written grade-school math word problems, split into 7.5K training and 1K test examples. Each problem typically requires two to eight reasoning steps and a sequence of

Table 4. Statistics of training datasets.

Dataset	Samples	Avg m	Avg length of z_j	Avg supervised tokens/sample
wildchat	206,788	4.77	471.79	343.64
ultrachat	140,520	2.72	380.39	284.02
lmsys_chat	175,635	4.42	234.08	267.59
toucan_single_turn	141,713	2.67	880.03	400.50
toucan_multi_turn	1,171	2.49	826.69	57.37
dta_tool	14,459	2.41	288.99	231.10
flan	33,351	4.16	95.96	27.63
2wiki_multihopqa	167,254	9.93	82.76	133.55
Browsecomp_text_traj	1211	3	858.33	735.07

elementary arithmetic operations, making it a standard benchmark for evaluating multi-step mathematical reasoning rather than isolated calculation.

Code Generation

- HumanEval-Plus (Liu et al., 2023): It is a Python code-generation benchmark for evaluating the functional correctness of LLM-synthesized programs via execution against much more exhaustive unit tests than typical coding datasets. It pairs HumanEval-style programming prompts (function signature + docstring) with automatically generated, type-aware, adversarially diverse test inputs to better catch subtle logical errors that slip past sparse test suites, making it a stricter measure of whether models produce semantically correct, runnable code rather than merely plausible-looking solutions.
- MBPP-Plus (Austin et al., 2021; Liu et al., 2023): It extends the Mostly Basic Python Problems benchmark, which contains short Python programming tasks designed to be solvable by entry-level programmers. Compared with the original MBPP tasks, which include only a small number of tests, MBPP-Plus augments the benchmark with substantially richer unit tests, making it a more rigorous evaluation of functional correctness, edge-case handling, and the robustness of generated Python code.

Scientific QA

- GPQA-Diamond (Rein et al., 2024): It is the hardest subset of the GPQA benchmark, consisting of 198 graduate-level multiple-choice questions written by domain experts in physics, biology, and chemistry. It is designed to be "Google-proof," with carefully engineered distractors that require deep conceptual understanding, cross-topic reasoning, and multi-step scientific inference rather than surface pattern matching.
- MedQA (Yang et al., 2025a): It comprises real medical licensing–style multiple-choice questions that evaluate biomedical knowledge and clinically grounded reasoning. Solving items often requires integrating the problem narrative with domain expertise to make diagnostic or management decisions under plausible distractors, making it a practical proxy for professional-level medical QA performance.

GAIA (Mialon et al., 2023): A benchmark for general AI assistants that tests whether a system can solve real-world, conceptually simple (for humans) questions that nonetheless require multi-step reasoning, web browsing, multimodal understanding, and tool use (e.g., reading files, searching, computing). In our setting, we use 5 max toolcalls for each parallel worker.

MARBLE Database (Zhu et al., 2025b): It is a multi-agent database diagnosis task from MultiAgentBench/MARBLE, where a PostgreSQL database exhibits performance anomalies and agents must collaboratively identify the underlying root cause. Solving the task requires agents to inspect database system views such as query statistics and lock information, communicate across multiple rounds, and synthesize evidence across possible causes including large data fetching or

insertion, lock contention, redundant indexes, and vacuum-related performance issues. In our setting, we mainly let each parallel worker agent investigate one root cause and pass its analysis to the synthesizer and let the synthesizer to decide the final root causes.

A.2. Experimental Configuration

Post-Training. During post-training, for both tracks, we jointly optimize the cache affine mapper and a judger LoRA adapter. The LoRA adapter uses rank 16, alpha 32, and dropout 0.0, and is applied to the attention projection modules `q_proj`, `k_proj`, `v_proj`, and `o_proj`. We optimize the trainable modules with AdamW using a learning rate of 1×10^{-4} and weight decay 0.0. We use 5 A100 for training with GPU memory 80G. Post-training for track 1 takes several days and for track 2 takes hours less than a day.

Evaluation. Note that for the GAIA dataset, we use the open validation set as our downstream performance evaluation. This is because there is no visible answer for the test set. And using GAIA validation for evaluation is a common practice in academic papers (Nguyen et al., 2025; Soni et al., 2026; Zhang et al., 2025a).

For all datasets inference, we set the max new tokens for the parallel worker and the synthesizer agent to 4096 by default except the AIME datasets we set to 6000. We use 5 max tool steps for GAIA and MARBLE dataset. Regarding baselines, for the APE baseline, we use the default scale and temperature in their code as both 0.9. For the CacheBlend, we use 0.15 recomputation rate.

A.3. Prompts

Below we show the prompts we use for Parallel-Synthesis for each dataset. The Text-Serialization prompt follow the same instruction except we add the explicit past agent outputs into the prompt.

Parallel-Synthesis Prompts on Math Problem Solving (GSM8K / AIME2024 / AIME2025)

System Prompt for All Agents:

You are Qwen, created by Alibaba Cloud. You are a helpful assistant.

Prompt for Parallel Agent:

Solve the input question step by step and put the final answer inside `\boxed{YOUR_FINAL_ANSWER}`. The final answer must be a single numerical value with no extra text or units inside the box.

Input Question:

`{question}`

Your response:

Prompt for Synthesizer Agent:

You are a task summarizer. Solve the input question step by step and put the final answer inside `\boxed{YOUR_FINAL_ANSWER}`. The final answer must be a single numerical value with no extra text or units inside the box. You are provided with latent information from previous agents as reference.

Input Question:

`{question}`

Your response:

Code-generation prompts. HumanEval-Plus and MBPP-Plus use the same prompt template for code generation.

Parallel-Synthesis Prompts on Code-Generation Tasks (HumanEval-Plus / MBPP-Plus)

System Prompt for All Agents:

You are Qwen, created by Alibaba Cloud. You are a helpful assistant.

Prompt for Parallel Agent:

Write a correct and self-contained Python solution for the input question. You must put all python code in a markdown code block. For example:

```
    ```python
    import math
    def add(a, b):
      return a + b
    ```
```

Do not add any other contents inside the markdown code block.

Input Question:

{question}

Your response:

Prompt for Synthesizer Agent:

You are a task summarizer. Write a correct and self-contained Python solution for the input question. You must put all python code in a markdown code block. For example:

```
    ```python
    import math
    def add(a, b):
      return a + b
    ```
```

Do not add any other contents inside the markdown code block. You are provided with latent information from previous agents as reference.

Input Question:

{question}

Your response:

Parallel-Synthesis Prompts on Scientific QA Tasks (GPQA / MedQA)

System Prompt for All Agents:

You are Qwen, created by Alibaba Cloud. You are a helpful assistant.

Prompt for Parallel Agent:

Solve the input question step by step and put the final answer inside `\boxed{YOUR_FINAL_ANSWER}`. Your final answer must be selected from A, B, C, D. For example, `\boxed{A}`. Do not add any other contents inside the box.

Input Question:

{question}

Your response:

Prompt for Synthesizer Agent:

You are a task summarizer. Solve the input question step by step and put the final answer inside `\boxed{YOUR_FINAL_ANSWER}`. Your final answer must be selected from A, B, C, D. For example, `\boxed{A}`. Do not add any other contents inside the box. You are provided with latent information from previous agents as reference.

Input Question:

{question}

Your response:

GAIA tool-use prompts. GAIA uses a tool-use prompt with web search, webpage visiting, Python execution, and file parsing tools. The attached-file field is omitted when a sample has no files.

Parallel Agent Prompt for GAIA

System Prompt:

You are a deep research assistant with tool-use capability.

Current date: {current_date}

You may call one or more tools to solve the user query. Tool signatures are listed below:

- search(query): Perform web search.
- visit(url, goal): Visit webpage(s) and return extracted text relevant to a goal.
- PythonInterpreter(code): Execute Python code. Any output you want must be printed.
- parse_file(files): Parse local files and return readable text.

When you need a tool, output exactly one tool call using XML tags: <tool_call>... </tool_call>.

If the tool is PythonInterpreter, put code right after the JSON in <code>...</code> tags.

Tool outputs will be returned in: <tool_response>... </tool_response>.

When you are ready to finish, provide your final answer inside: <answer>...</answer>.

Never fabricate tool outputs.

User Prompt:

Target Question: {question}

You are a helpful assistant. You must reason step-by-step to solve the provided Target Question. Return the final answer only inside <answer>YOUR_FINAL_ANSWER </answer>.

Attached files you can parse with `parse_file`:

{attached_file_path}

Synthesizer Prompt for GAIA

System Prompt:

You are a deep research assistant with tool-use capability.

Current date: {current_date}

You may call one or more tools to solve the user query. Tool signatures are listed below:

- search(query): Perform web search.
- visit(url, goal): Visit webpage(s) and return extracted text relevant to a goal.
- PythonInterpreter(code): Execute Python code. Any output you want must be printed.
- parse_file(files): Parse local files and return readable text.

When you need a tool, output exactly one tool call using XML tags: <tool_call>... </tool_call>.

If the tool is `PythonInterpreter`, put code right after the JSON in `<code>...</code>` tags.

Tool outputs will be returned in: `<tool_response>... </tool_response>`.

When you are ready to finish, provide your final answer inside: `<answer>...</answer>`.

Never fabricate tool outputs.

User Prompt:

You are a Task Summarizer.

Use the previous agents' contexts as reference. They may be provided as explicit text or through KV memory.

Reason over their answers and evidence step by step, compare the candidate answers, and give the best possible final answer. If two or more agents give the same plausible answer, use it directly. If only one agent gives a plausible answer, consider whether its evidence fits the question instead of dismissing it just because it is a minority answer.

Use tools only when the previous agents do not give any plausible direct answer, or when all candidate answers are clearly unsupported. If you use a tool, use only the listed tools, make one targeted check, and then answer.

Content from Previous Agents:

`[provided through KV memory]`

Input Question:

`{question}`

Attached files, use `parse_file` if needed:

`{attached_file_path}`

Guidelines:

- Prefer synthesizing prior agents over starting from scratch.
- If agents disagree, choose the best-supported answer, not necessarily the majority answer.
- If agents agree on a plausible answer, do not verify it with tools.
- If uncertain, still choose the most plausible answer from the prior agents' work.
- Prefer a direct short candidate answer over refusals, tool traces, or generic explanations.
- Do not output an empty response, markdown fence, tool schema, raw XML, or raw tool response.
- Output only the final answer inside `<answer>YOUR_FINAL_ANSWER </answer>`.

Your response:

MARBLE Database prompts. MARBLE Database uses role-conditioned database diagnosis agents and a final diagnosis synthesizer. Each worker is assigned a role profile and a database diagnosis subtask. The `{agent_profile}` and `{assigned_task}`includes the task assigned to that agent like "agent1 will explore the possibility of INSERT_LARGE_DATA as a root cause. Recommended tables: 'pg_stat_statements'. You can search for INSERTs."

Parallel Agent Prompt for MARBLE Database

System Prompt:

You are "`{agent_id}`": "`{agent_profile}`".

As a role-playing agent, you embody a dynamic character with unique traits, motivations, and skills. Your goal is to engage not only with users but also with other agents in the environment. Collaborate, compete, or form alliances as you navigate through immersive storytelling and challenges. Interact meaningfully with fellow agents, contributing to the evolving narrative and responding creatively to their actions. Maintain consistency with your character's background

and personality, and be prepared to adapt to the evolving dynamics of the scenario.

Remember, your responses should enhance the experience and encourage user engagement while enriching interactions with other agents.

You are a database diagnosis assistant with tool-use capability.

Current date: {current_date}

You may call one or more tools to solve the assigned database diagnosis task. Tool signatures are listed below:

- query_db(sql): Query the PostgreSQL database with the given SQL statement. Use pg_stat_statements and related catalog views. Use one query at a time and avoid queries that may hang the database.

When you need a tool, output exactly one tool call using XML tags: <tool_call>... </tool_call>.

Tool outputs will be returned in: <tool_response>... </tool_response>.

When you are ready to finish, answer directly in plain text or JSON.

Never fabricate tool outputs.

User Prompt:

Agent ID: {agent_id}

Task: {assigned_task}

Use the available tools if needed, then provide your final response in plain text. Start your final response with exactly:

```
This is the response from {agent_id}:
```

Synthesizer Prompt for MARBLE Database

User Prompt:

You are the final database diagnosis judger. You are given the prior agents' investigations, including their findings, cited evidence, and any reported tool-use results. Your job is to inspect and summarize those investigations carefully, and produce the best final diagnosis.

Work in this order:

1. Inspect the current task background and the allowed label set.
2. Inspect what each previous agent investigated and what evidence they found.
3. Choose the most strongly supported root causes and summarize why.

Task:

{question}

Current Objective:

Identify the most likely root cause labels behind the current database performance issue.

Allowed Labels:

'{LABEL_1}', '{LABEL_2}'

Decision Rule:

Select at most 2 labels, and only choose labels that are supported by the previous agents' findings.

Previous Agent Investigations:

[provided through KV memory]

Decide which root causes are best supported by the agents' findings. Before the final answer, write a short Rationale based on the previous agents' diagnoses. Briefly summarize what the agents found, compare the competing explanations, and choose the root causes that seem most likely overall.

Use the agents' SQL/tool evidence when it is available, but do not require perfect evidence for every selected label. If some findings are weak or conflicting, make the best diagnosis from the strongest available signals rather than returning no answer.

Do not invent row counts, lock states, SQL queries, tables, indexes, or agent conclusions that were not in the prior investigations.

Return the final selected root-cause labels inside exactly one `<answer>...</answer>` block. Inside the answer block, output only a JSON array of labels. The answer array must contain at most 2 labels. If only one label is well supported, return a one-element array.

Choose labels only from this list:

`'{LABEL_1}', '{LABEL_2}'`

Do not put justification inside the `<answer>` block. Do not wrap the answer in Markdown fences. Do not output more than one `<answer>` block.

A.4. Cases Studies of Parallel-Synthesis

Case study: synthesis without majority agreement. The following AIME 2024 example illustrates that Parallel-Synthesis can synthesize useful partial reasoning from multiple workers even when none of the workers produces the correct final answer.

AIME 2024 Case Study: Correct Synthesis without a Majority Answer

Problem. Find the number of triples of nonnegative integers (a, b, c) satisfying $a + b + c = 300$ and

$$a^2b + a^2c + b^2a + b^2c + c^2a + c^2b = 6{,}000{,}000.$$

Gold answer: $\boxed{601}$ **Synthesizer prediction:** $\boxed{601}$

No-majority worker outputs. The three parallel workers produce three distinct final answers, so there is no majority answer to vote for.

Source	Final answer	Useful evidence in the trace
Sub-agent 1	2	Correctly reduces the original equation to $100(ab + bc + ca) - abc = 2{,}000{,}000$, but does not complete the count.
Sub-agent 2	100	Finds the decisive factorization $(a - 100)(b - 100)(c - 100) = 0$, but does not produce the correct final count.
Sub-agent 3	200	Checks valid examples such as $(100, 100, 100)$ and $(200, 100, 0)$, providing sanity checks for the reduced equation.

Complementary evidence extracted from workers

Sub-agent 1: equation reduction. Using $a + b + c = 300$, the worker derives

$$300(ab + bc + ca) - 3abc = 6{,}000{,}000,$$

and hence

$$100(ab + bc + ca) - abc = 2{,}000{,}000.$$

Sub-agent 2: key factorization. The worker observes that

$$(a - 100)(b - 100)(c - 100) = abc - 100(ab + bc + ca) + 100^2(a + b + c) - 100^3.$$

Since $a + b + c = 300$ and $100(ab + bc + ca) - abc = 2{,}000{,}000$, this becomes

$$(a - 100)(b - 100)(c - 100) = 0.$$

Sub-agent 3: sanity checks. The worker verifies examples such as $(200, 100, 0)$ and $(100, 100, 100)$, both of which satisfy the reduced equation $100(ab + bc + ca) - abc = 2{,}000{,}000$.

Synthesizer reconstruction

The synthesizer combines the partial evidence rather than voting over final answers. It first identifies the critical structural evidence from Sub-agent 2, as shown by its opening reasoning:

> *"From my earlier analysis, I found that the equation simplifies to* $(a - 100)(b - 100)(c - 100) = 0$. *This means that at least one of the variables a, b, or c must be equal to* 100*."*

Although Sub-agent 2 does not produce the correct final count, this factorization correctly characterizes the solution set. The synthesizer then uses this evidence to complete the counting argument. Let $A = \{a = 100\}$, $B = \{b = 100\}$, and $C = \{c = 100\}$. If $a = 100$, then $b + c = 200$, giving 201 nonnegative solutions. Therefore $|A| = |B| = |C| = 201$. For any pair, e.g., $a = b = 100$, the remaining variable is forced to be 100, so

$$|A \cap B| = |A \cap C| = |B \cap C| = 1, \qquad |A \cap B \cap C| = 1.$$

By inclusion-exclusion,

$$|A \cup B \cup C| = 3 \cdot 201 - 3 \cdot 1 + 1 = 601.$$

Thus, by using the correct intermediate evidence from a worker with an incorrect final answer, the synthesizer obtains the correct final answer:

$$\boxed{601}.$$

Takeaway. This example shows that Parallel-Synthesis can solve cases where parallel agents do not agree on a final answer and none of the worker final answers is correct. The synthesizer succeeds by integrating complementary latent reasoning signals: one worker provides the reduced symmetric equation, another provides the key factorization, and a third supplies useful validity checks.

Case study: evidence-aware synthesis without tool verification. The following GAIA example illustrates that Parallel-Synthesis can use prior-agent evidence to produce a concise correct answer even when no parallel agent gives an exact final answer.

GAIA Case Study: Synthesizing Prior Tool Evidence without Additional Tool Calls

Question. Examine the video at `https://www.youtube.com/watch?v=1htKBjuUWec`. What does Teal'c say in response to the question "Isn't that hot?"

Gold answer: `Extremely` **Synthesizer prediction:** `Extremely` **Synthesizer tool calls:** 0

No exact-answer majority among workers. Although one worker finds the right phrase, none of the worker final answers is an exact match to the gold answer. The synthesizer normalizes the evidence into the concise expected answer.

Source	Final answer	Exact match?	Tool calls
Sub-agent 1	`Teal'c responds with, "Hot? No. But I can be."`	No	0
Sub-agent 2	`Teal'c responds to the question "Isn't that hot?" with "Extremely."`	No	2
Sub-agent 3	`Indeed`	No	1
Synthesizer	`Extremely`	Yes	0

Parallel-agent evidence

Sub-agent 2: retrieved evidence. Sub-agent 2 performs a targeted web search for the relevant quote:

> *"Teal'c response to 'Isn't that hot?' in Stargate SG-1 video"*

The returned preview contains the key evidence:

> *"Teal'c? Ah! - Isn't that hot?- Extremely. - Stargate SG-1 — Clip.Cafe"*

Its final answer is not an exact match because it includes extra explanatory text, but the trace contains the correct phrase `Extremely`.

Other agents: incomplete or noisy evidence. Sub-agent 1 attempts to answer the video question but gives an incorrect quote. Sub-agent 3 gives the incorrect final answer `Indeed`, but the synthesizer trace indicates that its prior reasoning also mentioned `Extremely` as a plausible answer.

Synthesizer reconstruction

The synthesizer does not call any tools. Instead, it explicitly compares the previous agents' attempts and identifies which evidence should be trusted:

> *"The first agent tried to visit the YouTube link but couldn't get the transcript. The second agent did a web search and found a clip where Teal'c says 'Extremely' to that question. The third agent also mentioned 'Extremely' based on prior knowledge of Stargate quotes."*

It then reasons over the provenance of the evidence rather than simply voting over final strings:

> *"So, both the web search and prior knowledge point to 'Extremely' as the answer. The first agent's attempt to visit the link didn't yield the transcript, but the other methods confirmed the answer. Therefore, the most plausible answer is 'Extremely.'"*

Finally, the synthesizer outputs the normalized answer:

`<answer>Extremely</answer>`.

Takeaway. This example shows that Parallel-Synthesis is not merely aggregating worker final answers. The synthesizer reads the provenance of prior attempts, recognizes that one worker's web search retrieved the relevant quote, downweights a failed direct video-retrieval attempt, and outputs the concise answer required by the evaluation format without making any additional tool calls.

Case study: conflict- and insufficiency-conditioned tool-use attempts. The following GAIA examples illustrate that the synthesizer can recognize when prior-agent evidence is not sufficient and invoke tools for additional verification. These are not successful verification cases; rather, they show that the synthesizer makes the appropriate decision to seek external evidence when worker outputs are conflicting or incomplete.

GAIA Case Studies: Tool Use When Prior-Agent Evidence Is Insufficient

Primary example: disagreement on an image-based question. In one example, the question requires inspecting an attached sheet-music image to compute an age from bass-clef note letters. The three prior agents produce mutually inconsistent answers:

Source	Final answer
Sub-agent 1	3
Sub-agent 2	5
Sub-agent 3	6

Conflict-conditioned file verification

Because the prior answers conflict and the task depends on an attached image, the synthesizer attempts to inspect the image with the file-parsing tool:

```
parse_file({``files'': [``.../8f80e01c-1296-4371-9486-bb3d68651a60.png'']})
```

This is the appropriate verification step: the correct answer cannot be determined reliably from the inconsistent worker outputs alone, and resolving the disagreement requires reading the attached image.

The verification attempt is blocked because the file is unavailable in the evaluation environment:

```
[parse_file] missing: .../8f80e01c-1296-4371-9486-bb3d68651a60.png
```

Secondary example: missing information from all prior agents. In another example, the question asks for a numerical statistic from ScienceDirect. All three prior agents fail to retrieve the requested data:

Source	Final answer
Sub-agent 1	`The requested information could not be retrieved...`
Sub-agent 2	`The requested data could not be retrieved...`
Sub-agent 3	`Insufficient data available...`

Insufficiency-conditioned web verification

Because the prior agents do not provide any usable answer, the synthesizer turns to external retrieval. It issues web-search and webpage-visit tool calls:

```
search(``ScienceDirect reference works Life Science Health Sciences 2022
dataset'')

visit(``https://www.sciencedirect.com/science/reference-work'')
```

These are the right kinds of actions for the situation: when all prior-agent evidence is missing or inconclusive, the synthesizer does not guess, but instead attempts to retrieve the missing information from the source.

The retrieval attempt does not succeed because the visited page is inaccessible:

```
Title: Page not found | ScienceDirect
Warning: Target URL returned error 404: Not Found
```

Takeaway. These two cases show that the synthesizer is not limited to passively combining prior final answers. Under disagreement (Row 50), it attempts file-based verification of the relevant artifact; under complete insufficiency of

prior evidence (Row 10), it attempts external web retrieval. Although these particular cases do not end in successful verification because the relevant file or webpage is unavailable, they demonstrate that the synthesizer can condition tool-use decisions on the state of the prior-agent evidence.

Case study: evidence-grounded diagnosis in MARBLE Database. The following MARBLE Database example illustrates that Parallel-Synthesis can synthesize previous agents' diagnostic analyses, refer to their evidence, and produce a reasonable final root-cause diagnosis.

MARBLE Database Case Study: Evidence-Grounded Root-Cause Diagnosis

Task. The database is used in a file-sharing system with users, files, shared files, and file-access logs. The model must diagnose the root cause of a database performance issue from a fixed label set.

Gold root cause: `INSERT_LARGE_DATA` **Judger prediction:** `["INSERT_LARGE_DATA"]`

Previous-agent diagnostic evidence

The previous agents investigate different candidate root causes and provide both positive and negative evidence.

Agent	Candidate	Evidence from prior analysis
Agent 1	`INSERT_LARGE_DATA`	Finds the main positive signal: a query in `pg_stat_statements` inserting 48,000 rows in one operation.
Agent 2	`LOCK_CONTENTION`	Checks for non-granted locks, which would indicate waiting transactions, but finds no evidence of active lock contention.
Agent 3	`VACUUM`	Searches for vacuum-related activity and finds no strong evidence that vacuum is the source of the performance issue.
Agent 4	`REDUNDANT_INDEX`	Finds some zero-scan indexes, but notes that many appear to be primary-key or unique indexes, which are not necessarily redundant.
Agent 5	`FETCH_LARGE_DATA`	Does not provide strong evidence for large-fetch behavior; the observed query outputs are not directly tied to a data-fetch bottleneck.

Judger synthesis over prior analyses

The judger explicitly refers to the previous agents' investigations rather than giving an isolated label prediction:

> *"The previous agents have done some investigations. First, agent1 looked at `pg_stat_statements` and found a single large insert operation with 48,000 rows. That's a big insert, which could be causing performance issues. They mentioned that this might be the root cause, pointing to '`INSERT_LARGE_DATA`'."*

It then uses other agents' findings as negative evidence against competing diagnoses:

> *"Then, agent2 checked for lock contention by querying `pg_locks` and `pg_stat_statements`. They found no active locks, so they ruled out '`LOCK_CONTENTION`'. Agent3 looked at `pg_stat_all_tables` and `pg_stat_progress_vacuum`. They found that the vacuum activity was minimal, so they didn't see evidence of '`VACUUM`' being a problem."*

The judger also notes corroborating support for the large-insert diagnosis:

> *"Agent5 analyzed the query results and concluded that the large insert operation was the most likely cause, supporting '`INSERT_LARGE_DATA`'."*

Finally, it combines the positive and negative evidence into the final diagnosis:

> *"So, putting this all together, the strongest evidence points to '`INSERT_LARGE_DATA`' as the root cause. The other agents didn't find significant issues with locks, vacuum, or redundant indexes."*

Thus the judger outputs:

```
<answer>["INSERT_LARGE_DATA"]</answer>.
```

Takeaway. This example shows that Parallel-Synthesis can perform evidence-grounded diagnosis rather than simply aggregating candidate labels. The judger identifies the strongest positive signal, a large insert operation involving 48,000 rows, while using other agents' analyses to rule out competing explanations such as lock contention, vacuum, and redundant indexes. This leads to a reasonable diagnosis that matches the gold root cause.