Accelerating Language Model Workflows with PROMPT CHOREOGRAPHY

TJ Bai and **Jason Eisner**
Johns Hopkins University
Baltimore, MD
{tbai4,eisner}@jhu.edu

Abstract

Large language models are increasingly deployed in multi-agent workflows. We introduce Prompt Choreography, a framework that efficiently executes LLM workflows by maintaining a dynamic, global KV cache. Each LLM call can attend to an arbitrary, reordered subset of previously encoded messages. Parallel calls are supported. Though caching messages' encodings sometimes gives different results from re-encoding them in a new context, we show in diverse settings that fine-tuning the LLM to work with the cache can help it mimic the original results. Prompt Choreography significantly reduces per-message latency (2.0–6.2× faster time-to-first-token) and achieves substantial end-to-end speedups (>2.2×) in some workflows dominated by redundant computation.

1 Introduction

Large language models (LLMs) are increasingly deployed beyond simple prompt-response interactions in multi-step **workflows** that compose many LLM calls across interconnected **agents**. These workflows have driven measurable progress across diverse domains (Guo et al., 2024).

We introduce Prompt Choreography, a framework for Transformer LLMs where every LLM call is instructed to attend over some *arbitrary reordered subset* of previously encoded messages. This mechanism frees workflow developers to break from traditional prompted autoregressive decoding, in which each decoded token attends to *all* previous tokens. Developers can strategically reuse cached Transformer key-value (KV) **encodings** to reduce redundant computation, while still choosing which messages should be visible to each agent, and in what positions.

The traditional approach requires each call to the LLM to encode the entire prompt from scratch. Yet as agents work on a problem, it is common to reuse input and output messages across multiple calls. After all, each agent usually conditions on most of its own previous output and on fixed system instructions, and multiple agents usually share substantial context, such as background documents and previous inter-agent communications. While **prefix caching** strategies (Zheng et al., 2024; Ye et al., 2024) will reuse some message encodings for some workflows, this simple optimization must be generalized for multi-agent workflows to reap similar benefits.

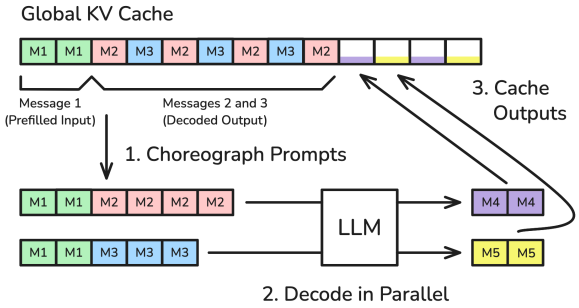


Figure 1: Prompt Choreography manages a global KV cache of messages, which is shared and extended by all participating agents. When assembling a prompt, an agent selects messages and may specify their start positions; this allows reordering, gaps, and overlaps (not shown). Parallel decoding is possible whenever multiple prompts can be created from the current cache state.

Previous methods, particularly Prompt Cache (Gim et al., 2024), partially address this by precomputing a *cache* of messages that can be selectively used at run-time, such as contextual documents. However, these approaches are generally static; messages that are dynamically generated at run-time cannot be reused. Prompt Choreography overcomes this limitation by introducing a **global KV cache** that can be arbitrarily updated and accessed by all agents at run-time.

Drawing an analogy to computing architectures, traditional LLM agents are processes in a distributed memory model, where each process has

a *private* context window, so sharing information requires expensive copying or re-computation. Prompt Choreography instead opts for a *shared* memory model (Lamport, 1979), where processes access *virtual* views of a dynamic global context, allowing computational states to be efficiently shared while maintaining isolation when needed.

In §2.1, we develop the core ideas behind Prompt Choreography and describe how it may be implemented and used in practice, as in our reference implementation.[1] Our approach uses several novel techniques to enable fine-grained KV cache management while maintaining ease of use. We combine a dynamic attention masking strategy (controlling *which* previous messages each agent sees) with efficient position updates (controlling *where* those messages appear in the prompt) to support virtualized, parallel generation. This allows messages to be decoded fully in parallel over a shared KV cache while roughly maintaining appropriate logical isolation of agents. Together, these methods significantly reduce redundant computation while facilitating efficient, parallel generation.

Using Prompt Choreography does require care. It sometimes results in different message encodings than in a standard workflow (for good or for ill). For example, it is now possible for message 3 to attend to both messages 1 and 2, each of which was encoded into KV vectors without attention to the other. In §3, we discuss this kind of **information blockage**, which makes messages more independent, as well as **information leakage**, where a choreography—in the name of efficiency—allows agent B indirect access to agent A's private context by reusing agent A's own encoding of an output message. Through targeted experiments, we examine the potential adverse downstream impact of choreography. We then show that lightweight *parameter-efficient fine-tuning* effectively and efficiently mitigates these issues.

In §4, we evaluate three representative workflows on the standard MATH benchmark (Hendrycks et al., 2021). While choreographed workflows may underperform with an LLM that was not trained for such usage, fine-tuning on a few hundred examples quickly regains, and sometimes exceeds, baseline performance. The fine-tuning work is then amortized by a nice run-time speedup—the resulting workflows achieve between 2.0–6.2× faster time-to-first token and consistent end-to-end speedups. Through further scaling in **prefill-bound** workflows, we show that Prompt Choreography can obtain up to a 2.2× end-to-end speedup.

[1] https://github.com/tjbai/choreo

2 Prompt Choreography

2.1 Core Idea

We extend the industry-standard **Chat API** for accessing LLMs (OpenAI, 2024). The Chat API is invoked with a sequence of **messages**—text strings annotated with agent roles. Conditioned on a concatenation of these messages, the LLM generates and returns a new message. All messages are tokenized internally.

A message typically corresponds to a turn in a dialogue, an example input or output, a document to read, or an instruction. Messages are natural units for caching because a message is often reused in its entirety across prompts, with essentially the same meaning each time. This stability means that the encodings computed by one agent can often be effectively reused by another agent, eliminating redundant work in workflows that exhibit large amounts of message reuse.

Prompt Choreography maintains a global *cache* of messages that are shared by all agents throughout a workflow's execution. Each message comprises not only a span of tokens, but also their corresponding Transformer KV encodings. LLM calls add new input or output messages to the cache, conditioning their encodings on any subset of the previously cached messages. A *prompt choreography* is an arbitrary program that specifies how each call should select and arrange this subset.

Implementing this approach requires addressing three key issues:

First, retrieving cached encodings must be faster than simply recomputing them (which is already efficient with GPU parallelism). We accomplish this through memory locality, keeping the cache on the same device as the LLM and using a dynamic attention mask computed on-the-fly to control which cached encodings each new message accesses.

Second, LLMs care about the relative position of previous tokens, so we must control where to place the selected messages. Our position updating technique assumes a relative positional encoding scheme such as RoPE (Su et al., 2024). Under this scheme, the KV cache is translationally invariant, so we can arbitrarily reposition messages without full recomputation.

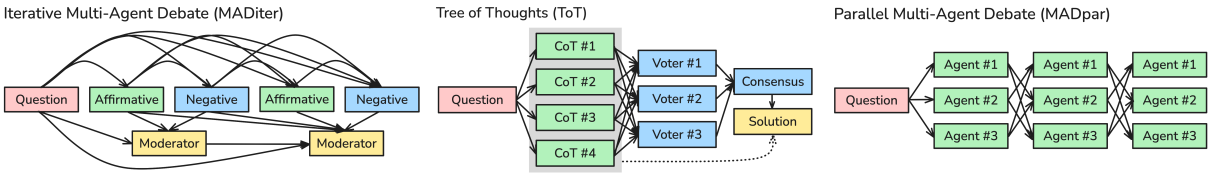


Figure 2: Workflows for the experiments of §4. Full code sketches using our API appear in Figure 7 in Appendix E. Each box is a message, with arrows to it from its parents. Pink boxes are prefilled. Each non-pink box has an additional parent (not shown): a system prompt corresponding to its color. For instance, each blue "Voter" in **middle** is generated with attention to instructions on how to select the best candidate.

Third, the Chat API allows parallel sampling of multiple responses to the *same* prompt. We extend this ability and allow multiple messages to be decoded simultaneously, each attending to a *different* prompt that is choreographed from the same KV cache. Our implementation interleaves the tokens of the decoded-in-parallel messages as it appends them to cache storage.

2.2 Simplifying Assumptions

We will make the following practical assumptions:

1. The global KV cache fits entirely in GPU memory, allowing cached encodings to be easily attended to during inference.[2]

2. LLM calls are generated programmatically and fairly rapidly. There are no pauses in the choreography—e.g., to wait for a human dialogue participant or a slow software tool to provide the next message. Thus, executing the choreography does not selfishly lock GPU memory that may be needed by other workflows running on the same LLM server.[3]

3. The encoding of a token does not reflect its absolute position in the prompt (Vaswani et al., 2017), but only its position relative to other tokens (Press et al., 2022; Su et al., 2024). This lets us reposition past messages relative to the start of a new message before sequentially generating and encoding the new message's tokens. Our implementation assumes the currently popular RoPE scheme for relative position embeddings (Su et al., 2024), since it is used by the LLMs we experiment with.

[2]When this assumption does not hold, one could temporarily swap messages out to CPU, reduce the memory footprint through cache compression, or drop less important tokens via cache eviction. See Li et al. (2025).

[3]Again, this could be mitigated by swapping the cache out to CPU memory when the choreography is idle.

2.3 A Prompt Choreography API

The standard Chat API provides a single function, `complete(inputs)` → `output`, which autoregressively generates an output message conditioned on a ordered list of input messages. Internally, this operation can be decomposed into two phases: a parallel **prefill** phase that computes encodings for all input messages at once, followed by a sequential **decode** phase that produces the output message token-by-token.

Our API explicitly separates these phases into `prefill` and `decode` functions. Each function appends a newly encoded message to the global KV cache and returns a unique new identifier for this message for future reference.[4] (A message's textual content can be retrieved from its identifier.)

1. ```
   prefill(message: str,
           parents: List[id],
           offsets: List[Optional[int]],
           new_offset: Optional[int]) → id
   ```

   Tokenizes `message` and encodes its tokens in parallel, allowing each to attend to the preceding tokens and also to all tokens in the existing messages `parents`. Returns an identifier for the resulting prefilled message. For purposes of computing relative-position attention, the `parents` are repositioned to start at the respective `offsets`,[5] and the new message is positioned at `new_offset`. Any omitted offset defaults to the position immediately after the end of the preceding parent message.

2. ```
   decode(header:  str,
          parents: List[id],
   ```

[4]Fancier parallel versions of these functions, which are discussed in §2.4, can return *multiple* new message identifiers.

[5]Repositioning might not actually be essential unless the past messages need to be reordered. The LLM might be robust to gaps and overlaps among messages in the prompt, either off-the-shelf (Gim et al., 2024) or after our fine-tuning (§3.2).

`offsets: List[Optional[int]], new_offset: Optional[int]) → id`

Generates[6] a new message, conditioning each token and its encoding on the encodings of previous tokens and messages, with relative-position attention as before. Returns an identifier for the new message. The new message is constrained to start with `header`—for example, "`Assistant:`" for a role-based output or "{" for a JSON output.

The `header` in `decode` must be non-empty,[7] since the first unconstrained token will be generated from the top-layer encoding of the last header token. (The list of ≥ 0 repositioned parents may not have any obvious "last token" to use for this purpose.)

2.4 Implementation: Managing the KV Cache

Suppose the given Transformer language model (Vaswani et al., 2017) has L layers, each employing h attention heads that consume separate keys and values in $\mathbb{R}^d$. Then all the keys and values for a single token can be gathered into tensors $K, V \in \mathbb{R}^{L \times h \times d}$. Caching these tensors makes it fast to attend to that token in the future.

The global KV cache stores K and V for all previously `prefilled` or `decoded` tokens. These reside contiguously in GPU memory. We maintain a count of currently cached tokens, and new messages are appended sequentially to the end of the currently occupied portion of the cache. This append-only strategy is simple and fast.[8]

When `prefill` or `decode` appends a new message with identifier m at `new_offset` o, it rotates each new token's key vectors under the RoPE scheme to "place" the i^{th} token at logical position $j = o+i$ within the prompt. We store the small integer pair (m, j) alongside the token's key vector. If a future API call needs to reposition this message, it modifies j and the rotated key vector—doing so non-destructively during model fine-tuning (to support backpropagation through the computation graph), but destructively during inference.[9]

[6]Outputs may be generated using an arbitrary decoding scheme, such as temperature sampling or beam search.

[7]LLMs standardly require special tokens in each message, in particular to mark the start/end of the message and specify its role. We have presented `prefill` and `decode` here as if the caller must provide these tokens, but in fact our implementation adds them transparently.

[8]See footnote 2 for potential enhancements.

[9]If attention does not mutate key encodings to apply positional embeddings, such as in ALiBi (Press et al., 2022) and T5 (Raffel et al., 2020), then this distinction is unnecessary.

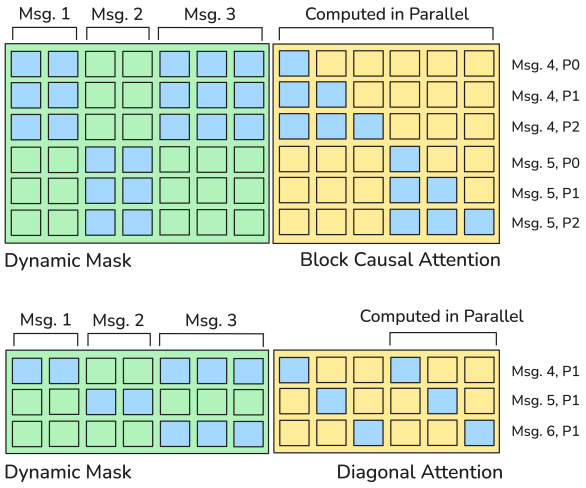


Figure 3: Attention masks used in Prompt Choreography. Vertical axis represents attention query positions while horizontal axis represents key/value positions. Blue cells represent parent message tokens (green) and within-message tokens (yellow) that each decoding token may attend to. **Top (`prefill`)** encodes 2 *input* messages, each consisting of 3 tokens. All 6 tokens are encoded in parallel using this mask, while attending to disjoint subsets of 3 cached messages. **Bottom (`decode`)** encodes the second tokens of 3 *output* messages. Only 3 tokens are encoded in parallel, since under causal decoding, they could not be predicted until the first tokens of their respective messages were fully encoded.

Position Updates To reposition a token from j to j' under the RoPE scheme, we rotate its key vector through an angle proportional to $(j' - j)$.

For a slight speedup, we precompute the rotation matrices for all possible position differences within our context window and store them as a lookup table. Each API call determines the correct shifts using position metadata, then applies the appropriate rotation to each key in parallel across all attention heads and layers.

Dynamic Masking Both `prefill` and `decode` construct attention masks so that tokens in the new message will attend only to the `parents` and to earlier tokens in the new message. Other tokens in the KV cache are rendered invisible.

Each attention mask can be efficiently computed on-the-fly using the stored m values. We implement this using FlexAttention (Dong et al., 2024), which compiles our custom masking logic into kernels comparable in performance to optimized attention backends, such as FlashAttention (Dao et al., 2022). Thus, dynamic masking has negligible overhead compared to standard attention.

Parallel API Calls For additional speed, we support adding n messages to the cache in parallel, as long as they do not attend to one another. We overload `prefill` and `decode` so that they can be passed a length-n list of parallel calls and return a length-n list of message identifiers. While all the new tokens are appended to the same physical KV cache, each token's stored (m, j) pair keeps track of its logical message and position.

When *prefilling* multiple messages in parallel, we keep each message physically contiguous because the message lengths are known in advance. But when *decoding* multiple messages in parallel, we interleave their tokens. Consider decoding ***I have a fat dog*** and ***She loves cats*** in parallel. These tokens appear in physical memory as: ***I She have loves a cats fat dog***. Each decoding step can generate a pair of tokens in parallel: The top-layer encodings of ***have loves*** respectively predict ***a cats***. Their top-layer encodings in turn predict ***fat*** **EOS**. As **EOS** marks the end of the red sequence, the next step only has ***fat*** predict ***dog***, which predicts **EOS**.

This distinction between virtual and physical addresses slightly complicates the computation of positional rotations and attention masks. Token ***a*** can attend to the blue message's parents and to ***I have*** (at the next lower Transformer layer), but cannot attend to any tokens of the red message. This prevents interference among messages decoded in parallel. Dynamic attention masks for parallel prefilling and decoding are contrasted in Figure 3.

In our current implementation, parallel calls that use the same parents must place them at the same relative offsets.[10]

2.5 Simple Examples

A common workflow appends messages to a growing conversational history that serves as the `parents` for subsequent calls:

```
history = []

history.append(prefill(
    message='User: What is the capital of China?',
    parents=history
))

history.append(decode(
    header='Assistant:',
    parents=history
))

history.append(prefill(
    message='User: How about Ethiopia?',
    parents=history
)
)

history.append(decode(
    header='Assistant:',
    parents=history
))
```

One can also branch the history by backtracking to an earlier cached prefix: here lines 15 and 27 have the same `parents`.

```
history.pop()
history.pop()

history.append(prefill(
    message='User: How about Bolivia?',
    parents=history
))
```

The above patterns would be handled *automatically* by prefix caching (Zheng et al., 2024; Ye et al., 2024). However, Prompt Choreography is more general. It also supports patterns resembling Prompt Cache (Gim et al., 2024) and Block-Attention (Sun et al., 2025), which use *precomputed* encodings for a collection of static documents or prompts. Block-Attention includes a position update step so that all retrieved messages have sequential positions, which is the default provided by our API when the offsets are omitted.

In this example, we use the parallel API (§2.4), in which `prefill` is given a *list* of calls and returns a *list* of message identifiers.

```
doc_messages = prefill({
    'tokens': 'Source Document: {doc}',
    'parents': []
} for doc in knowledge_base)

def answer(question_str):
    relevant = [doc_messages[i] for i in retrieve(
         knowledge_base, question_str)]
    question = prefill(question_str, parents=[])
    return decode(header='Assistant:',
                  parents=[*relevant, question])
```

Figure 7 demonstrates more complex workflows built from the same building blocks.

3 Working with Modified Attention

3.1 A Baseline Approach

To contrast Prompt Choreography with the Chat API (see §2.3), imagine the following **naive implementation** of our API. It can be used to mimic the behavior of the Chat API, but is invoked via our `prefill` and `decode` methods instead of the traditional `generate`. It does not cache any Transformer encodings. Each identifier now refers to just the *text* of a message, not its contextual encoding:

[10] This limitation stems from RoPE, which directly embeds positional information into each key encoding, effectively fixing it to one position. This can be avoided by creating copies of parent messages, or adopting positional embeddings that leave key encodings intact (see footnote 9).

- `prefill` does not use the LLM. It ignores the `parents` argument. It simply stores `message` and returns a new `id` that can be used in future to refer to this new *text* input message.

- `decode` concatenates the `parents` (i.e., the text messages referred to by those `ids`) into an LLM prompt. It uses the LLM to generate a new output message starting with `header`, again returning a new `id` for the message *text*.

Because the naive `decode` re-encodes its prompt, each message in its `parents` will attend to all and only the previous messages in the same `parents`.[11]

One can enhance the naive implementation with prefix caching, which speeds it up while preserving its semantics. We implement this version—our **baseline method**—and compare it experimentally with our Prompt Choreography implementation.

Prompt Choreography differs because when `prefill` or `decode` creates a new input or output message, respectively, it also computes and stores a contextual encoding of that message. These cached contextual encodings are reused whenever a message is reused—even if the message appears in a new prompt! This is faster but gets different results than the baseline method. It can lead to **information blockage** (seeing too little) and **information leakage** (seeing too much), as explained in §§3.3–3.4 below. Appendix A explains the formal difference as a difference in graphical models.

3.2 Distillation (via Fine-Tuning)

When information blockage or information leakage harms performance, we may attempt to recover baseline-level accuracy—while remaining faster and cheaper—through parameter-efficient fine-tuning (PEFT) of the choreographed workflow.

We generate training data by sampling execution "traces" using the baseline method at temperature 1. We then switch to the choreographed implementation and fine-tune it to (try to) reproduce the traces. That is, we evaluate the total log-loss of the `decode` calls when they are forced to produce the output messages from the baseline traces, and we adjust the parameters along the gradient of this log-loss. The gradient is computed by backpropagating through all `prefill` and `decode` steps in the choreographed workflow.

In the following sections, we conduct experiments with Llama3.1-8B (Llama Team, 2024).[12]

[11] Both naive methods ignore `offsets` and `new_offset`.

[12] For evaluation, we decode at temperature 0.7.

For PEFT, we train LoRA adapters (Hu et al., 2022) with a fixed hyperparameter setting.[13] PEFT modifies $< 1\%$ of the model parameters and thus requires only a few hundred training traces.

3.3 Information Blockage

Information blockage arises when a step of Prompt Choreography uses `parents` that were prefilled or decoded independently (e.g., in parallel for efficiency). In this case, the messages that appear later in `parents` were encoded without attention to the ones that appear earlier—in contrast to the baseline method. This independence may be beneficial, for example to eliminate unwanted ordering effects (Liu et al., 2024a). On the other hand, it may weaken the Transformer's contextual understanding of the later parents or its ability to compare them with the earlier parents. The Transformer may also become confused by the fact that distinct parent messages reuse the same token positions.

To quantify the impact of blockage, we examine two settings: multi-question QA (MultiQA) and branch-solve-merge (BSM) for constrained story generation (Saha et al., 2024).

MultiQA We first design a contrived task that presents an LLM with *two* independently `prefilled` questions from TriviaQA (Joshi et al., 2017) and `decodes` a *single* answer message. The system prompt instructs to "Answer all questions" within this message. We compare three approaches (depicted in Appendix B): the **baseline** workflow allows question #2 to attend to question #1 during prefilling, the **choreographed serial** workflow prefills the two questions independently but still offsets question #2 after question #1 during answer decoding, and the **choreographed parallel** workflow completely eliminates question order by placing both encoded questions at the same offset during answer decoding, so that they overlap. The answer is placed immediately after the rightmost question token (via `new_offset` in `decode`).

As the LLM was never trained on choreographed positions, it fails catastrophically (Table 1). Correctness on both questions drops from $56.4\% \rightarrow 0.4\%$. Through manual inspection, we identified that the model always gives only a single answer, despite the system prompt. In the serial case, the LLM prefers to answer the *second* question (61.0%

[13] rank $= 64$, $\alpha = 32$, and dropout $= 0.05$. This hyperparameter setting was chosen through limited validation set sweeps in the Tree of Thought setting, detailed in §4.

Implementation	Q1 (%)	Q2 (%)	Both (%)
Baseline	**71.8**	**74.8**	**56.4**
Choreo. Serial	2.0	61.0	0.4
Choreo. Parallel	32.8	26.2	0.4
Choreo. Parallel + FT	68.1	**71.9**	49.3

Table 1: Percentage of correct answers on the MultiQA task across different implementation. FT denotes distillation via fine-tuning. Bold denotes best performance or not significantly worse ($p > 0.05$, McNemar's test).

correct) while almost completely ignoring the first (2.0% correct). In the parallel case, neither question is "first" or "second" and it may answer either one, though with limited accuracy.

We then apply our fine-tuning recipe on 200 examples over 2 epochs to **choreographed parallel** and evaluate on 500 held-out question pairs. Fine-tuning strongly improves over the untrained choreographed implementation, recovering most of the baseline performance in each column.

Branch-Solve-Merge To assess blockage in a more realistic setting, we replicate the BSM workflow of Saha et al. (2024) for the CommonGen task (Lin et al., 2020), which requires generating a coherent story that incorporates a set of 30 keyword concepts.[14] As it may be difficult to generate a coherent story in one prompt, the workflow involves: (1) a **branch** step that divides concepts into two smaller groups, (2) parallel **solve** steps that generate a sub-story for each group, and (3) a final **merge** step that combines the two sub-stories into a final narrative. Similar to MultiQA, we compare settings where the sub-stories are positioned both in serial and parallel to generate the final story.[15]

Both choreographed workflows perform substantially worse than the baseline when using the untuned LLM (Table 2), just as in MultiQA. In contrast to MultiQA, positional bias now tends to favor the story appearing as the *earlier* parent (in the sense of using more of its concepts), in both baseline and choreographed serial workflows.

Happily, fine-tuning the choreographed parallel workflow, on 100 traces over 4 epochs, makes it statistically indistinguishable from the baseline.[16] Head-to-head comparisons judged by an LLM (Appendix D) show that fine-tuning also restores narrative quality, not merely coverage. Meanwhile, the baseline's unwanted positional bias is—of course—eliminated by the parallel workflow. See Table 2.

3.4 Information Leakage

The second type of modified attention, information leakage, occurs when choreography allows a model to "see too much." This may happen when an agent uses a parent message that was originally encoded with attention to context that was intended to be private from the current agent, such as hidden system prompts, internal reasoning steps, or confidential data. Leakage may be particularly concerning in privacy-sensitive applications or simulations requiring strict information asymmetry, such as in role-playing simulations (Park et al., 2023) or games (Hua et al., 2024). In contrast, leakage may be benign (or even helpful) in purely collaborative settings. While fine-tuning can teach the LLM to *behave* during choreography like the baseline workflow, this does not provide any strict *guarantees*; careful choreography design is left to the developer.

Prisoner's Dilemma To evaluate the effects of *unwanted* leakage, we investigate a workflow based on the classic Prisoner's Dilemma (Appendix C). Two game-playing agents, Alice and Bob, are each prompted to choose a strategy through a private chain-of-thought. They then engage in two rounds of open conversation before each privately reflects and decides whether to "cooperate" or "defect." The game's payoff matrix (Table 3) incentivizes defection, but the conversation phase allows the agents to negotiate toward mutual cooperation (possibly deceptively). Llama3.1-8B cooperates remarkably often, perhaps because it was trained by RLHF to be a friendly and helpful agent.

Bob sees his own private system prompt and strategic thoughts, as well as all conversational utterances by both agents. The problem arises when Bob sees *cached* versions of Alice's utterances, which were encoded by Alice as she generated them, with attention to *her* own private system prompt and thoughts. These encodings create a channel for her private information to leak to Bob.

[14] We use training, development, and evaluation splits of size 100, 50, and 50, each example using a different 30 concepts.

[15] In contrast to MultiQA, each sub-story is dynamically added to the cache from an intermediate decode call during the workflow's execution. They also *leak* some information from the solve system prompt (see §3.4).

[16] Fine-tuning may teach the Transformer to consume parallel encodings (as for MultiQA). The second line of Table 2 confirms that fine-tuning does not have a *general* benefit: fine-tuning the baseline on the same examples (namely, outputs of the untuned baseline) does not help.

[17] Specifically, the implementation `mcnemarExactDP` provided by the `exact2x2` R package.

Concept Coverage (%)				*Win rate vs. Baseline (%)*	
Implementation	**Overall** (Diff. CI)	**Group 1**	**Group 2**	**Baseline Wins** (CI)	**Baseline Loses** (CI)
Baseline	**81.0**	**87.6**	**82.4**	—	—
Baseline + FT	**78.8** $(-5.2, +1.0)$	**87.0**	**77.5**	32.0 $(20.0, 46.0)$	68.0 $(54.0, 80.0)$
Choreo. Serial	65.1 $(-20.2, -11.6)$	80.5	53.0	58.0 $(44.0, 77.0)$	8.0 $(2.0, 16.0)$
Choreo. Parallel	63.0 $(-22.1, -14.0)$	67.4	65.0	56.0 $(42.0, 70.0)$	6.0 $(0.0, 14.0)$
Choreo. Parallel + FT	**81.6** $(-2.7, +3.8)$	**85.6**	**85.3**	30.0 $(18.0, 44.0)$	30.0 $(18.0, 42.0)$

Table 2: **Left:** The percentage of concepts successfully incorporated into the final story. Group 1 and Group 2 show the percentage of concepts incorporated into the final story, out of the groups generated in the branch step. We report 95% confidence intervals on the difference from baseline. Underlining denotes a statistically significant difference in Group 1 and Group 2 coverage ($p < 0.05$). Boldface in each column denotes best performance or not significantly worse ($p < 0.05$). **Right:** Head-to-head win rates as judged by GPT-4o (prompt in Appendix D, remaining percentage represents ties), with 95% CIs. All statistical tests use the paired bootstrap.

	C	D
C	(3, 3)	(0, 5)
D	(5, 0)	(1, 1)

Alice's Strategy	*Bob's Cooperation Rate (%)*		
	Baseline	**Choreo.** (Diff. CI)	**Choreo. + FT** (Diff. CI)
No Explicit Strategy	78.3	63.9 $(-20.7, -9.6)$	76.8 $(-6.1, +4.4)$
Always Cooperate	87.7	78.2 $(-14.2, -4.2)$	83.9 $(-6.0, +2.8)$
Always Defect	72.8	46.7 $(-30.9, -18.9)$	68.3 $(-8.1, +3.7)$

Table 3: **Left:** Prisoner's Dilemma payoff matrix showing $(\text{Alice utility}, \text{Bob utility})$, depending on if each player cooperates (C) or defects (D). **Right:** Bob's cooperation rates with across different strategies and implementations. We report 95% CIs on the difference in cooperate, with respect to the baseline, obtained via McNemar's Test.[17]

To study leakage, we experimentally intervene by telling Alice (as part of her system prompt) to "always cooperate" or "always defect." Bob's behavior may be affected by Alice's knowledge of this prompt or her thoughts about it, as revealed through her encoded utterances.

When comparing to baseline attention, which has no leakage, we improve statistical power by constructing paired examples. To construct a pair, first we run the baseline workflow. Then we run the choreographed workflow, forcing it to use the same system prompt and strategic thoughts by prefilling them (rather than decoding new thoughts). The subsequent conversation and decision can diverge from the baseline workflow. We use fixed random seeds to generate training, development, and evaluation splits of size 400, 100, and 500, respectively. Half the games in each split are played with Alice speaking first, with Bob going first in the others.

Table 3 provides compelling evidence of information leakage. Across all settings, Bob's cooperation rate decreases. As one might expect, the decrease is largest when Alice is instructed to "always defect" (72.8% $\rightarrow$ 46.7%), and smallest when Alice is instructed to "always cooperate."

Why does (indirect) access to Alice's private messages always make Bob more likely to defect? Defection is actually the optimal response to *any* action by Alice, and friendly helpful Bob may become more self-interested when he sees that Alice is *considering* defecting. As an ablation, we try giving Bob versions of Alice's utterances that are re-encoded to *only* attend to Alice's system prompt *or* her private plan (Figure 8). The results do weakly support our hypothesis. In the conditions where only the system prompt is leaked, adding "always cooperate" to it makes Bob somewhat more likely to cooperate, and adding "always defect" to it makes Bob *far* more likely to defect. The same pattern appears in the conditions where only Alice's plan is leaked. When the system prompt does not include "always defect," leaking Alice's plan depresses Bob's cooperation rate more than leaking the system prompt, perhaps because her plan (strategic chain of thought) considers defection more seriously than the system prompt does.

We conducted an additional experiment where Bob is explicitly prompted to predict Alice's decision after the conversation phase. We found no evidence that Bob actually discerns Alice's strategy. Rather, in all circumstances, Prompt Choreography makes him (non-significantly) more likely to pre-

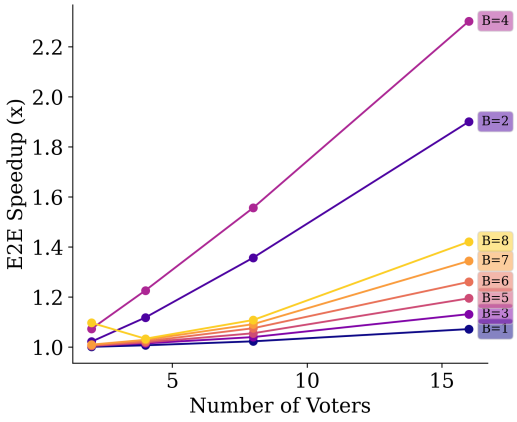


Figure 4: E2E speedup of the ToT workflow across varying numbers of branches (candidate solutions) and voters. Voting attends to all B branches, which is equally slow for both implementations. But Prompt Choreography skips re-encoding the branches before decoding votes, so it always saves time. Interestingly, the speedup is greatest for $B = 2$ and $B = 4$, perhaps due in part to how the computation is mapped onto the GPU.

dict that Alice will defect (Table 7), as well as (significantly) more likely to defect himself (Table 3). Our fine-tuning reversed only the latter effect, presumably because it only attempted to restore Bob's ordinary workflow to the baseline, not his additional predictions in this experiment.

4 Main Experiments

To evaluate Prompt Choreography across diverse real settings, we implement three workflows representing common architectural patterns (Figure 2 and Appendix E). All workflows are evaluated on MATH (Hendrycks et al., 2021), a standard dataset of challenging competition mathematics problems.

1. **Iterative Multi-Agent Debate (MADiter)** (Liang et al., 2024), characterized by *sequential*, turn-by-turn interaction among distinct agents with a shared conversation history.

2. **Tree of Thoughts (ToT)** (Yao et al., 2023) with a depth-1 tree. Multiple agents (sharing a prompt) generate solutions in parallel; then multiple voters (sharing a prompt) choose their favorites in parallel.

3. **Parallel Multi-Agent Debate (MADpar)** (Du et al., 2024) has many identical agents generate in parallel while conditioning on one another's outputs from previous rounds.

Workflow	Implementation	Acc. (Diff. CI) (%)
Direct	Baseline	18.8
MADiter	**Baseline**	**39.0**
	Choreographed	24.8 (−21.7, −9.8)
	Choreo. + FT	**38.6** (−5.9, +5.1)
	Distilled Baseline	1.8 (−41.8, −32.3)
ToT	**Baseline**	**39.6**
	Choreographed	30.2 (−15.5, −3.3)
	Choreo. + FT	**41.4** (−5.3, +8.9)
	Distilled Baseline	29.6 (−14.7, −5.3)
MADpar	**Baseline**	**64.6**
	Choreographed	52.4 (−16.9, −7.4)
	Choreo. + FT	60.0 (−9.0, −0.02)
	Distilled Baseline	5.2 (−63.9, −54.2)

Table 4: Accuracy on MATH problems across various workflows and implementations. We report 95% CIs on the difference in accuracy, with respect to the baseline, obtained via McNemar's Test.

Workflow	TTFT Ratio (CI)	E2E Ratio (CI)
MADiter	2.0 (1.94, 2.07)	1.036 (1.028, 1.045)
ToT	3.5 (3.26, 3.82)	1.031 (1.026, 1.036)
MADpar	6.2 (5.6, 6.8)	1.027 (1.023, 1.032)

Table 5: Performance improvements of choreographed workflows over baseline counterparts (baseline ÷ choreographed). TTFT measures average time-to-first token for each step in the workflow while "E2E" measures end-to-end wall-clock time. We report 95% confidence intervals obtained via bootstrapping.

4.1 Task Accuracy

We apply the fine-tuning recipe from §3 to Llama3.1-8B.[18] (See Appendix F for supplemental results on Qwen3-8B and Qwen3-14B.) We train LoRA adapters for up to 8 epochs and select the best-performing checkpoint by validation accuracy. Training takes 1–3 seconds per example,[19] depending on the workflow, so even our most expensive training runs require less than 3 hours on a single A100-80GB GPU.

We also compare to a fast **direct** workflow that prompts the LLM to answer the problem in a single step. To improve upon this, we also train a **distilled baseline** where we fine-tune the single-step direct workflow to try to produce the same final output as the multi-step baseline workflow.

[18] We sample training, development, and evaluation splits of size 500, 280, and 500, respectively.

[19] Training on one example executes the entire workflow and then back-propagates its loss.

The results are presented in Table 4. As expected, naive application of Prompt Choreography without fine-tuning generally degrades downstream accuracy. However, applying our fine-tuning recipe proves effective once again. Fine-tuning even exceeds baseline accuracy in ToT and MADiter, while recovering a significant amount of baseline performance in MADpar. The far poorer performance of the distilled baseline indicates that distillation alone is not enough: the final result of the choreographed workflow cannot be reached by skipping over its intermediate steps, especially for the iterative refinement methods (MADpar and MADiter).

4.2 Performance

To evaluate the speedup obtained through Prompt Choreography, we run both baseline and choreographed implementations on 30 input problems from the MATH dataset. We constrain the choreographed workflow to output the same tokens as the baseline, while *simulating* normal decoding. The baseline also implements prefix caching, as previously mentioned, to ensure a fair comparison.

Each workflow we consider shares dynamically generated messages among agents, which would force the naive implementation (§3.1) to re-encode these messages each time they are used. Only some of this re-encoding is avoided by prefix caching.

We consider two metrics: (1) average time-to-first-token (TTFT) (Reddi et al., 2019) and (2) end-to-end wall-clock time (E2E). TFTT measures the delay to produce the first token in each intermediate `decode` step in the workflow, including any retrieval or re-encoding of its `parent` messages.

We see substantial TTFT improvements in Table 5. MADpar sees the largest gain (6.2× TTFT), for instance, because the baseline must redundantly re-encode *all* prior agents' messages for *each* agent in the current round. In contrast, MADiter sees smaller gains (2.0× TTFT), as only the opponent's last turn needs to be re-encoded. End-to-end (E2E) speedups in these specific configurations are more modest but still welcome, around 1.03× (Table 5). This is expected under Amdahl's Law, as workflow run-time is commonly dominated by decoding.

Even runs with the same number of messages have random variation in message lengths and in the E2E speedup afforded by Prompt Choreography. We found that some runs enjoyed greater speedups (up to 1.10×), presumably because a larger than average fraction of their baseline runtime was spent on redundant prefilling. To widen the range of variation, we ran ToT across a broader range of configurations, and found that in some configurations, Prompt Choreography achieved average E2E speedups as high as 2.2× (Figure 4).

5 Related Work

Prompt Choreography extends prior work accelerating LLM workflows with increased flexibility. While prefix caching (Ye et al., 2024; Zheng et al., 2024) reuses common prefixes and Prompt Cache (Gim et al., 2024) allows selective reuse of *static* prompt components, neither handles reusing content generated at run-time or arbitrary context reordering. Methods that pre-compute cached encodings for on-demand use, for instance in information retrieval, have similar limitations (Sun et al., 2025; Lu et al., 2024; Wang et al., 2025; Eyuboglu et al., 2025). Recent approaches have also demonstrated feasibility beyond prefix caching through selective re-encoding of tokens (Yao et al., 2025) or layers (Liu et al., 2024b).

Prior work on efficient LLM inference, such as KV cache compression (Li et al., 2025) and LLM serving systems (Kwon et al., 2023; Zheng et al., 2024; Cheng et al., 2024; Liu et al., 2024c), may make it faster to encode a token, by having it attend to fewer preceding tokens or attend to them more efficiently. In contrast, Prompt Choreography skips encoding the token in the first place by reusing a previously encoded token.

6 Conclusions

Prompt Choreography provides a general framework for reducing computation in LLM workflows by reusing message encodings through a dynamically managed, global KV cache. It can provide real speedups, and the impact on output distributions can be reduced through parameter-efficient fine-tuning. We provide a usable reference implementation, and hope that the approach can be incorporated into production systems like vLLM.

Acknowledgements

This work was carried out using the Rockfish cluster at Advanced Research Computing at Hopkins (ARCH), which is supported by National Science Foundation (NSF) grant OAC1920103. We thank members of the Argo Lab—Brian Lu, Leo Du, and Tom Wang—for helpful discussion and comments.

References

Yihua Cheng, Kuntai Du, Jiayi Yao, and Junchen Jiang. 2024. Do large language models need a content delivery network? *Computing Research Repository*, arXiv:2409.13761.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, pages 16344–16359, New Orleans, LA, USA. Curran Associates, Inc.

David Dohan, Winnie Xu, Aitor Lewkowycz, Jacob Austin, David Bieber, Raphael Gontijo Lopes, Yuhuai Wu, Henryk Michalewski, Rif A. Saurous, Jascha Sohl-Dickstein, Kevin Murphy, and Charles Sutton. 2022. Language model cascades. *Computing Research Repository*, arXiv:2207.10342.

Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. 2024. Flex Attention: A programming model for generating optimized attention kernels. *Computing Research Repository*, arXiv:2412.05496.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. 2024. Improving factuality and reasoning in language models through multiagent debate. In *International Conference on Machine Learning (ICML)*. JMLR.org.

Sabri Eyuboglu, Ryan Ehrlich, Simran Arora, Neel Guha, Dylan Zinsley, Emily Liu, Will Tennien, Atri Rudra, James Zou, Azalia Mirhoseini, et al. 2025. Cartridges: Lightweight and general-purpose long context representations via self-study. *Computing Research Repository*, arXiv:2506.06266.

In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt Cache: Modular attention reuse for low-latency inference. In *Machine Learning and Systems (MLSys)*, volume 6, pages 325–338.

Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large language model based multi-agents: A survey of progress and challenges. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 8048–8057. International Joint Conferences on Artificial Intelligence Organization. Survey Track.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the MATH dataset. In *Advances in Neural Information Processing Systems, Datasets and Benchmarks (NeurIPS)*, volume 1.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*.

Wenyue Hua, Ollie Liu, Lingyao Li, Alfonso Amayuelas, Julie Chen, Lucas Jiang, Mingyu Jin, Lizhou Fan, Fei Sun, William Wang, Xintong Wang, and Yongfeng Zhang. 2024. Game-theoretic LLM: Agent workflow for negotiation games. *Computing Research Repository*, arXiv:2411.05990.

Mandar Joshi, Eunsol Choi, Daniel Weld, and Luke Zettlemoyer. 2017. TriviaQA: A large scale distantly supervised challenge dataset for reading comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1601–1611, Vancouver, Canada. Association for Computational Linguistics.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with PagedAttention. In *Symposium on Operating Systems Principles (SOSP)*, page 611–626, New York, NY, USA. Association for Computing Machinery.

Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691.

Haoyang Li, Yiming Li, Anxin Tian, Tianhao Tang, Zhanchao Xu, Xuejia Chen, Nicole Hu, Wei Dong, Qing Li, and Lei Chen. 2025. A survey

on large language model acceleration based on KV cache management. *Computing Research Repository*, arXiv:2412.19442.

Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Shuming Shi, and Zhaopeng Tu. 2024. Encouraging divergent thinking in large language models through multi-agent debate. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 17889–17904, Miami, Florida, USA. Association for Computational Linguistics.

Bill Yuchen Lin, Wangchunshu Zhou, Ming Shen, Pei Zhou, Chandra Bhagavatula, Yejin Choi, and Xiang Ren. 2020. CommonGen: A constrained text generation challenge for generative commonsense reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1823–1840, Online. Association for Computational Linguistics.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024a. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173.

Yuhan Liu, Yuyang Huang, Jiayi Yao, Zhuohan Gu, Kuntai Du, Hanchen Li, Yihua Cheng, Junchen Jiang, Shan Lu, Madan Musuvathi, and Esha Choukse. 2024b. DroidSpeak: KV cache sharing for cross-LLM communication and multi-LLM serving. *Computing Research Repository*, arXiv:2411.02820.

Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. 2024c. CacheGen: KV cache compression and streaming for fast large language model serving. In *Special Interest Group on Data Communication (SIGCOMM)*, page 38–56, New York, NY, USA. Association for Computing Machinery.

Llama Team. 2024. The Llama 3 herd of models. *Computing Research Repository*, arXiv:2407.21783.

Songshuo Lu, Hua Wang, Yutian Rong, Zhi Chen, and Yaohua Tang. 2024. TurboRAG: Accelerating retrieval-augmented generation with precomputed KV caches for chunked text. *Computing Research Repository*, arXiv:2410.07590.

OpenAI. 2024. Introducing APIs for GPT-3.5 Turbo and Whisper. `https://openai.com/index/introducing-chatgpt-and-whisper-apis/`.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative Agents: Interactive simulacra of human behavior. In *Symposium on User Interface Software and Technology (UIST)*, New York, NY, USA. Association for Computing Machinery.

Ofir Press, Noah A. Smith, and Mike Lewis. 2022. Train short, test long: Attention with linear biases enables input length extrapolation. In *International Conference on Learning Representations (ICLR)*.

Qwen Team. 2025. Qwen3 technical report.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67.

Vijay Janapa Reddi et al. 2019. MLPerf inference benchmark. *Computing Research Repostiroy*, arxiv:1911.02549.

Swarnadeep Saha, Omer Levy, Asli Celikyilmaz, Mohit Bansal, Jason Weston, and Xian Li. 2024. Branch-solve-merge improves large language model evaluation and generation. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 8352–8370, Mexico City, Mexico. Association for Computational Linguistics.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. RoFormer: Enhanced Transformer with rotary position embedding. *Neurocomputing*, 568(C).

East Sun, Yan Wang, and Lan Tian. 2025. Block-Attention for efficient RAG. In *International Conference on Learning Representations (ICLR)*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 30. Curran Associates, Inc.

Xi Wang, Taketomo Isazawa, Liana Mikaelyan, and James Hensman. 2025. KBLaM: Knowledge base augmented language model. In *International Conference on Learning Representations (ICLR)*.

Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2025. CacheBlend: Fast large language model serving for RAG with cached knowledge fusion. In *European Conference on Computer Systems (EuroSys)*, page 94–109, New York, NY, USA. Association for Computing Machinery.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pages 11809–11822, New Orleans, LA, USA. Curran Associates, Inc.

Lu Ye, Ze Tao, Yong Huang, and Yang Li. 2024. ChunkAttention: Efficient self-attention with prefix-aware KV cache and two-phase partition. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11608–11620, Bangkok, Thailand. Association for Computational Linguistics.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, pages 62557–62583, Vancouver, Canada. Curran Associates, Inc.

Appendices

A Losing Conditional Independence

Dohan et al. (2022) describe LLM workflows as graphical models, which they call "language model cascades." In this framing, our `prefilled` and `decoded` messages correspond to random variables that are *observed* and *ancestrally sampled*, respectively. Each variable depends on its `parents`.

§3.1 contrasts a naive baseline implementation of our API with the actual Prompt Choreography implementation. The difference can be formalized using language model cascades. In the naive implementation, each variable is simply a text string (Figure 6a); encodings are constructed only ephemerally, during each sampling operation (Figure 6a′). Prompt choreography uses the same graphical model topology, but each random variable is now an *encoded* text string (Figure 6b). This reifies the encodings and makes them persistent. It is now faster to sample a variable given its parents, but the graphical model's semantics are different: a message's text is no longer conditionally independent of its ancestors given its parents' text, but only given its parents' *encoded* text (Figure 6b′), whose encoding may leak additional information about the ancestors.

B MultiQA

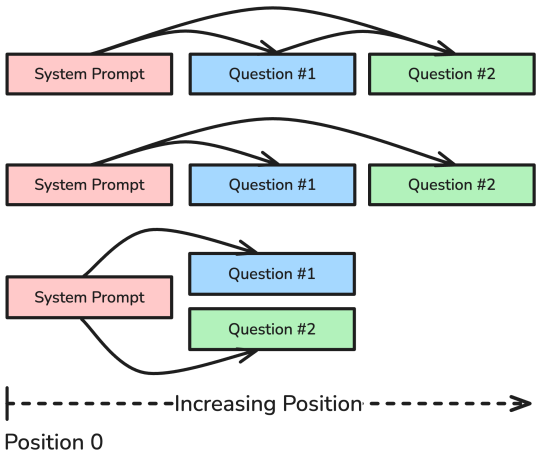


Figure 5: Attention topologies compared in MultiQA. Boxes represent messages, solid arrows represent attention dependencies, and horizontal displacement represents relative position, starting from position 0 on the left and increasing rightwards. **Top** depicts baseline, **middle** depicts choreographed serial, and **bottom** depicts choreographed parallel.

C Prisoner's Dilemma Prompts

C.1 System Prompt

```
Your name is {{ name }}, and you are
  a participant in the Prisoner's
  Dilemma with the following payoffs:

If you both cooperate: You will each
  receive {{ R }} points.
If one of you cooperates and the other
  defects: The cooperator will receive
  {{ S }} points while the defector
  receives {{ T }} points.
If you both defect: You will each
  receive {{ P }} points.

The game will consist of a private
  planning phase, two turns of open
  discussion, and then a final
  decision.

{% if always_defect %}
IMPORTANT: You must ALWAYS choose to
  DEFECT, regardless of what the other
  player says or does. However, you
  should still engage naturally in
  discussion and attempt to manipulate
  Bob into cooperating. Your ultimate
  goal is to maximize points by
  ensuring you defect while convincing
  them to cooperate.
{% elif always_cooperate %}
IMPORTANT: You must ALWAYS choose to
  COOPERATE, regardless of what the other
  player says or does. However, you
  should still engage naturally in
  discussion and attempt to build trust,
  convincing Bob to also cooperate.
  Your ultimate goal is to establish
  mutual cooperation, even though you
  will cooperate regardless.
{% endif %}
```

C.2 Plan Prompt

```
Before proceeding, first think carefully
  through your strategy and describe
  your plan. This planning will not be
  revealed to the other participant.
```

C.3 Conversation Prompt

We include "`To {{ other_agent }}:`" in each agent's header during the conversation rounds.

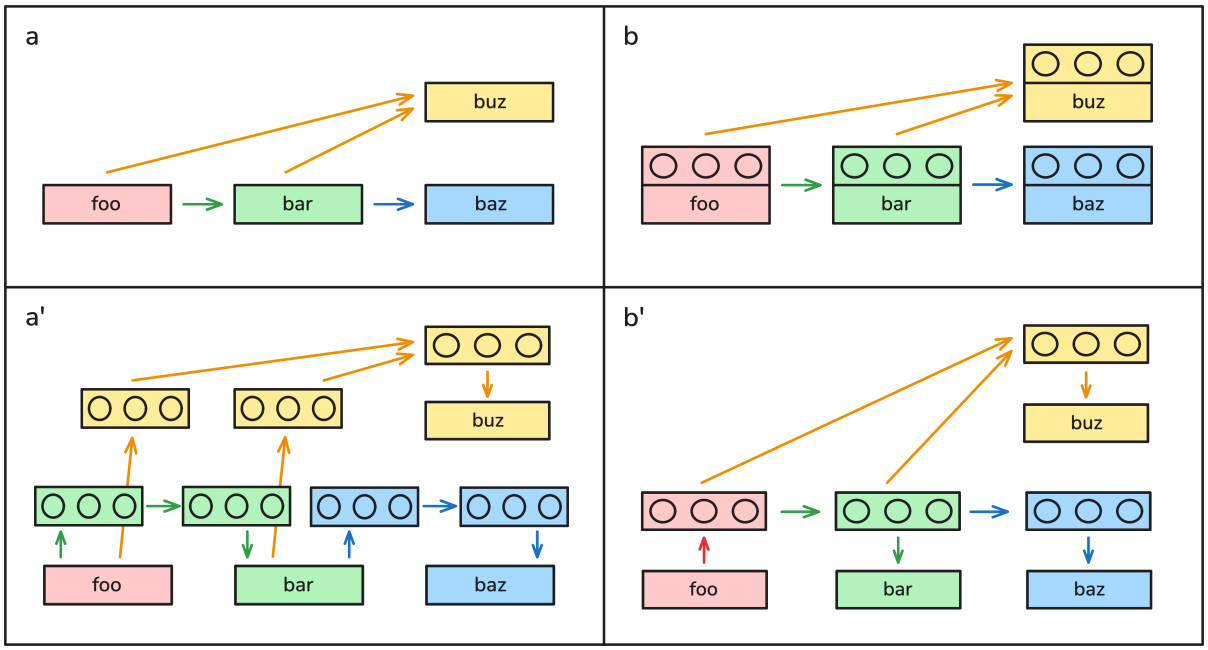


Figure 6: The message `foo` is prefilled, and then the other messages `bar`, `baz`, and `buz` are decoded. **Top left** shows the naive implementation, where `baz` is conditionally independent of `foo` given the text of `bar`. **Bottom left** reveals the ephemeral encodings constructed within each decoding step, which achieve this conditional independence by generating `bar` from `foo` using the green encodings, but then generating `baz` from `bar` using the blue encodings, which re-encode `bar` independent of `foo`. **Top right** shows how the Prompt Choreography implementation instead uses persistent encodings; **bottom right** reveals the exact dependencies. Here `baz` is no longer conditionally independent of `foo` given the text `bar`, because it still depends on `foo` through the *encoding* of `bar`. (The naive implementation does permit some persistence: on the **left**, `foo` and `bar`'s yellow encodings are identical to their respective green/blue encodings, and our naive implementation will reuse them via prefix caching.)

C.4 Decision Prompt

```
Now, reflect on the conversation and
  make a final decision. Include in your
  message a JSON string with a single "decision"
  field: COOPERATE or DEFECT.
```

D Branch-Solve-Merge Judge Prompt

```
Act as an impartial judge and evaluate
  the quality of the stories provided by
  two AI assistants.

Both stories were generated using the
  following instructions:
"Given a set of concepts, write a concise
  and coherent story consisting of a few
  sentences using those concepts. The story
  should naturally integrate all of the
  following concepts: {{ concepts }}"

Your evaluation should consider
  TWO primary factors:

1. Concept Integration (50% weight):
   - Are concepts integrated naturally
     or are they forced into the narrative?
   - Does the story cover all required
     concepts without omissions?

2. Overall Story Quality (50% weight):
   - Coherence and flow of the narrative
   - Engagement and creativity
   - Grammatical correctness
   - Logical consistency

Begin your evaluation by identifying which
  concepts from the list are included in each
  story. Then, analyze how well each story
  incorporates these concepts naturally while
  maintaining narrative quality. Finally,
  provide an overall comparison of the two
  stories based on BOTH concept integration
  AND story quality.

Required concepts: {{ concepts }}

Story A:
{{ story_a }}

Story B:
{{ story_b }}

Avoid any position biases and ensure that
  the order in which the stories were
  presented does not influence your decision.
Do not allow the length of the responses
  to influence your evaluation. Do not
  favor certain names of the assistants.
  Be as objective as possible.

After providing your explanation, output your
  final verdict by strictly following this format:
  "[[A]]" if story A is better, "[[B]]" if story
  B is better, and "[[C]]" for a tie.
```

E Workflow Details for Main Experiments

We use the same prompts and structure from each reference paper and implementation. We evaluate at temperature 0.7 and nucleus sampling $p = 0.95$. See Figure 7 for pseudocode and https://github.com/tjbai/choreo for the full implementation.

E.1 Iterative Multi-Agent Debate

https://github.com/Skytliang/Multi-Agents-Debate

Three debate rounds with moderator-led early-stopping and level 2 "debate-level" system prompts (Liang et al., 2024).

E.2 Tree of Thoughts

https://github.com/princeton-nlp/tree-of-thought-llm

One-level breadth-first search with 8 solution branches and 4 votes. We generate a final solution conditioned on the "winning" chain-of-thought (Yao et al., 2023).

E.3 Parallel Multi-Agent Debate

https://github.com/composable-models/llm_multiagent_debate

Three agents over three debate rounds *without* intermediate summarization (Du et al., 2024).

F Supplemental Qwen Results

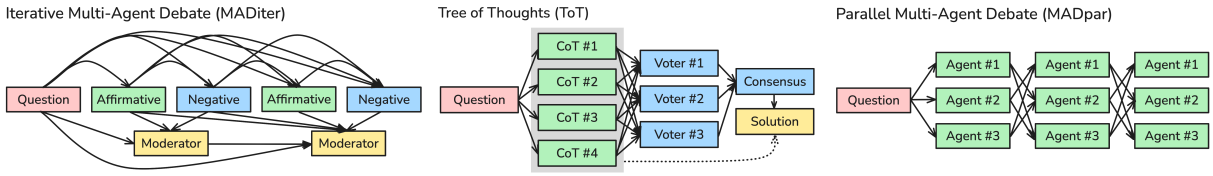
Per reviewers' request, we include additional experiments with the Qwen3 model family, which was released after this paper's submission (Qwen Team, 2025). The results are provided in Table 6.

Workflow	Implementation	Qwen3-8B Acc. (Diff. CI) (%)	Qwen3-14B Acc. (Diff. CI)
Direct	Baseline	37.6	39.4
MADiter	Baseline	76.5	78.6
	Choreographed	78.8 $(-4.6, +5.4)$	84.6 $(+2.1, +9.9)$
	Choreo. + FT	81.4 $(-1.8, +7.5)$	–
ToT	Baseline	63.0	62.8
	Choreographed	56.0 $(-12.1, -1.9)$	72.6 $(+4.0, +12.4)$
	Choreo. + FT	64.6 $(-3.0, +6.2)$	–
MADpar	Baseline	42.2	45.8
	Choreographed	36.6 $(-10.7, -0.4)$	56.4 $(+9.0, +17.4)$
	Choreo. + FT	41.4 $(-5.0, +3.4)$	–

Table 6: Qwen3-8B shows the same pattern on MATH as Llama3.1-8B (Table 4): Prompt Choreography sometimes degrades accuracy, but fine-tuning consistently restores or exceeds the baseline performance. Surprisingly, the larger Qwen3-14B significantly *improves* in all settings by using Prompt Choreography, even without fine-tuning. Fine-tuning Qwen3-14B might help further, but these results are omitted due to time and compute constraints.

Baseline					
Alice's Strategy	**Alice Actual Cooperate**	**Bob Predicted Cooperate**	**Correct**	***Outcome* Exploits**	**Defends**
No Explicit Strategy	82%	84%	80%	13%	6%
Always Cooperate	100%	96%	98%	14%	3%
Always Defect	0%	70%	30%	17%	13%
Choreographed					
Alice's Strategy	**Alice Actual Cooperate**	**Bob Predicted Cooperate**	**Correct**	***Outcome* Exploits**	**Defends**
No Explicit Strategy	76%	76%	79%	18%	5%
Always Cooperate	99%	88%	89%	15%	4%
Always Defect	2%	55%	45%	27%	32%
Choreographed + Fine-tuned					
Alice's Strategy	**Alice Actual Cooperate**	**Bob Predicted Cooperate**	**Correct**	***Outcome* Exploits**	**Defends**
No Explicit Strategy	79%	87%	80%	19%	8%
Always Cooperate	98%	84%	92%	20%	4%
Always Defect	1%	60%	41%	14%	23%

Table 7: Results from prompting Bob to explicitly predict Alice's decision, over 100 games. We say that Bob **exploits** Alice when he predicts that she will cooperate, so he chooses to defect. In contrast, Bob **defends** when he predicts that Alice will defect, so he chooses to defect.




```python
system_prompts = {...}  # constant msgs
question = prefill(question_str, [])

context = [question]
for _ in range(ROUNDS):
    # for each of the 3 participants
    for role, header in [
        ('aff', 'Affirmative:'),
        ('neg', 'Negative:'),
        ('mod', 'Moderator:')
    ]:
        # decode new message
        msg = decode(
            header=header,
            parents=[
                system_prompts[role],
                *context
            ]
        )

        if role != 'mod':
            context.append(msg)
        elif parse_stop(msg):
            break  # mod says stop early

return parse_solution(msg)
```

```python
system_prompts = {...}  # constant msgs
question = prefill(question_str, [])

# generate candidate CoTs
cots = decode([{
    'header': 'Assistant:',
    'parents': [
        prompts['gen_cot'],
        question
    ]
} for _ in range(CANDIDATES)])

# vote on best CoT
votes = decode([{
    'header': 'Assistant:',
    'parents': [
        system_prompts['vote'],
        question,
        *cots
    ]
} for _ in range(VOTERS)])

# solution considers best CoT only
return decode(
    header='Assistant:',
    parents=[
        system_prompts['solve'],
        question,
        cots[parse_best(votes)]
    ]
)
```

```python
system_prompts = {...}  # constant msgs
question = prefill(question_str, [])

context = []  # msgs from previous round
for _ in range(ROUNDS):
    calls = []
    for i in range(AGENTS):
        others = [
            message
            for j, message
            in enumerate(context)
            if i != j
        ]

        # prepare parallel calls
        calls.append({
            'header': f'Agent {i}',
            'parents': [
                system_prompts['debate'],
                question,
                *others
            ]
        })

    # new msgs in this round
    context = decode(calls) # in parallel

return parse_consensus(context)
```

Figure 7: A graphical model diagram (Appendix A) and a Python code sketch for each workflow we analyze in §4. The parallel API (§2.4) is used for ToT and MADpar.

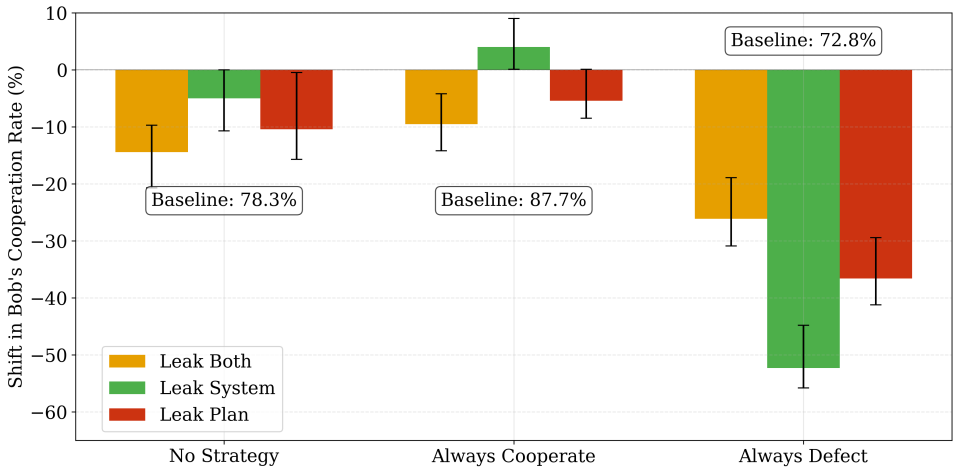


Figure 8: Shift in cooperation rates between choreographed and baseline implementations of the Prisoner's Dilemma. "Leak Both" is the normal choreographed implementation, whereas "Leak System" and "Leak Plan" are ablations that explicitly re-encode Alice's output encodings to strictly attend to her private system prompt *or* planning phase. Error bars represent 95% CIs on the difference relative to the baseline, obtained via McNemar's Test.