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-step workflows. We introduce Prompt Choreography, a framework that enables efficient execution of 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. While 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 latency (2.0–6.2x faster time-to-first-token) and achieves substantial end-to-end speedups (>2.2x) in workflows dominated by redundant computation.

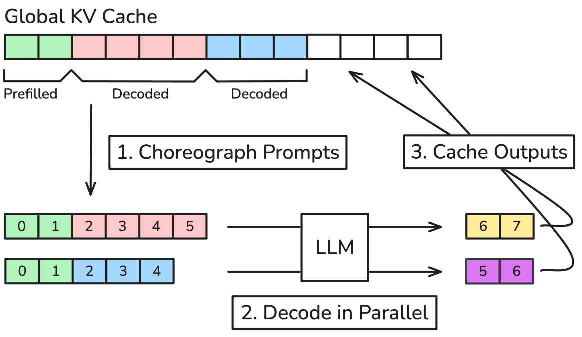


Figure 1: Prompt Choreography manages a global KV cache of messages, which is shared and modified by all participating agents. When assembling a prompt, the start position of each selected message may be specified by the user; this allows reordering, gaps, and overlaps (not shown here).

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 generated with attention over an *arbitrary reordered subset* of previously encoded messages. This mechanism gives worfklow developers freedom to break from traditional prompted autoregressive decoding, in which each decoded token attends to *all* previous tokens. By selectively determining which messages should be visible to each agent, and in what positions, developers can strategically reuse cached Transformer key-value (KV) **encodings** to reduce redundant computation.

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 single-agent workflows, this simple optimization must be generalized for multi-agent workflows to reap similar benefits.

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 effectively reused. Prompt Choreography overcomes these limitations 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 workflows operate in a distributed memory model, where individual agents have *pri-*

vate context windows that require expensive copying or re-computation to share information. Prompt Choreography instead opts for a *shared* memory model, where agents access **virtual** views of a global memory, 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, which forms the basis of our reference Python 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 context) 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 is encoded without attention to the other. In Section §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 an agent indirect access to another agent's private context by reusing that agent's own encoding of its output message. Through targeted experiments, we characterize the potential adverse downstream impact of choreography. Despite this, we 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 2.0–6.2x 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 2.2x 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 their *internal* meaning remains relatively stable even when the *external* context changes. 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 challenges:

First, retrieving cached values 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, although attention is fundamentally unordered, LLMs incorporate positional information into the context, so we must control where to place the selected messages. Our position updating technique exploits relative positional embedding schemes, such as RoPE (Su et al., 2023), to arbitrarily reposition messages without full recomputation.

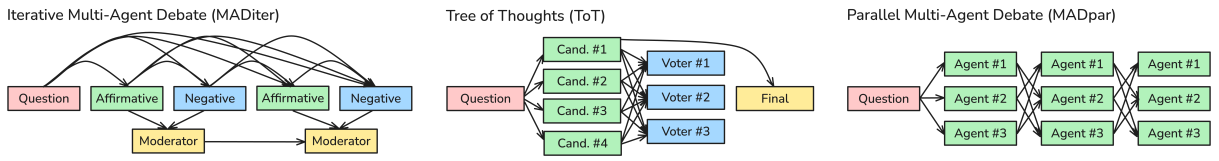


Figure 2: Workflows for the experiments of §4. Full pseudocode using our API appears in Figure 6 in the Appendix. Each box is a message, with arrows to it from its parents. Pink boxes are prefilled. Each non-pink box has an additional parent (not shown): a system prompt corresponding to its color. For instance, each blue "Voter" in **middle** is generated with attention to instructions on how to select the best candidate.

Third, to maintain the excellent parallel generation capability of LLMs, we combine the prior techniques with a special interleaved token layout and attention masking strategy. This allows multiple messages to be decoded simultaneously, each attending to a uniquely choreographed prompt, while sharing the *same* KV cache.

2.2 Simplifying Assumptions

Our implementation makes 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., 2023). 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., 2023), since it is used by the LLM 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., 2024).

[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, `generate(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 computes encodings that are added to the global KV cache and returns a unique message identifier for future reference.

1.
```
prefill(tokens: List[token],
        parents: List[id],
        offsets: List[Optional[int]],
        new_offset: Optional[int]) → id
```
   Encodes the given `tokens` in parallel, allowing each token to attend to the preceding given 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`,[4] 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: List[token],
       parents: List[id],
       offsets: List[Optional[int]],
       new_offset: Optional[int]) → id
```
   Generates and encodes tokens sequentially, conditioning on previous tokens and messages

[4]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).

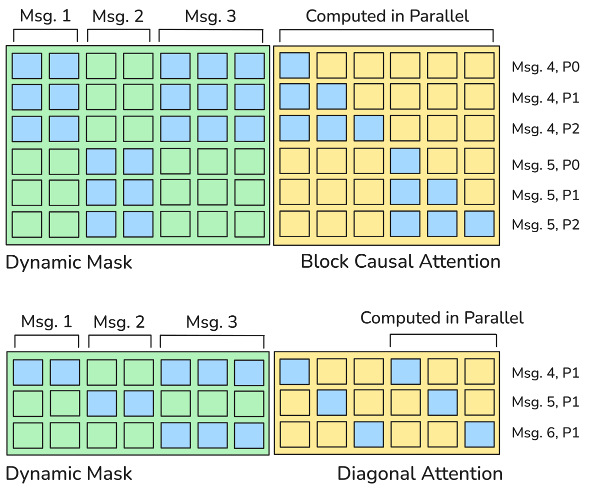


Figure 3: Attention masks used in Prompt Choreography. Each token of a new message corresponds to a row, whose blue entries indicate the parent message tokens (green) and new message tokens (yellow) that this token can attend to. **Top (`prefill`)** encodes 2 *input* messages, each consisting of 3 tokens. All 6 tokens are encoded in parallel using this mask. **Bottom (`decode`)** encodes the second tokens of 3 *output* messages. Only 3 tokens are encoded in parallel, since under Transformer decoding, they could not be predicted until the first tokens of their respective messages were fully encoded.

with relative-position attention as before. Returns an identifier for the newly decoded 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, since the first unconstrained token will be generated from the top-layer encoding of the last header token. (The collection of 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 `prefill`ed or `decode`d 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.[5]

When `prefill` or `decode` appends a new message with identifier m at `new_offset` i, it rotates each new token's key vectors under the RoPE scheme to "place" this token at logical position i within the prompt. We store the small integer pair (m, i) alongside the token's key and value vectors. If a future API call needs to reposition this message, we modify i and the rotated key vectors—non-destructively during model fine-tuning (to support backpropagation through the computation graph), but destructively during inference.

Dynamic Masking To control which cached encodings are attended to, we implement dynamic attention masking. Conceptually, when executing an LLM call with specific parent messages, we want to create a selective view of the global KV cache that includes only the relevant positions. We accomplish this by constructing an attention mask where each position is visible (unmasked) iff it belongs to either (1) the new message being created or (2) any of the parent messages.

This mask can be efficiently computed on-the-fly using the stored m values. We can equivalently define the mask using FlexAttention (Dong et al., 2024), which avoids materializing the entire mask and takes advantage of attention sparsity, allowing it to be competitive with efficient implementations such as FlashAttention (Dao et al., 2022). Thus, this operation has negligible overhead compared to standard attention.

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

For efficiency, 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.

Parallel API Calls For additional speed, we support adding multiple messages to the cache in parallel, as long as they do not attend to one another. While all the new tokens are appended to the same physical KV cache, each token's stored (m, i) pair keeps track of its logical message and position.

When *prefilling* multiple messages in parallel, we keep each message physically contiguous be-

[5]See footnote 2 for potential enhancements.

cause the message lengths are known in advance. But when *decoding* multiple messages in parallel, we alternate their tokens. Consider decoding "I have a dog" and "She loves her cat" in parallel. These tokens appear in physical memory as: "I She have loves a her dog cat." In each decoding step, we can generate a pair of tokens in parallel, so "dog cat" are predicted from the top-level encodings of from "a her" (respectively).

This slightly complicates the computation of attention masks. For example, "a" can attend to the blue message's parents and to "I have a" (at the next lower Transformer layer), but cannot attend to any tokens of the red message. This functions as a form of virtualization—each agent may choreograph prompts independently without concern for others, all while sharing the same global KV cache. Dynamic attention masks for parallel prefilling and decoding are contrasted in Figure 3.

2.5 Simple Examples

Basic scenarios like prefix caching, where subsequent calls reuse an initial conversational history, are naturally expressed by passing the id's of preceding messages in the `parents` list:

```
prefix = []

prefix.append(prefill(
    tokens='User: What is the capital of France?',
    parents=[]
))

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

q1 = prefill(
    header='User: How about Germany?',
    parents=prefix
)

q2 = prefill(
    header='User: How about China?',
    parents=prefix
)
```

While dedicated systems can efficiently handle *automatic* prefix caching (Zheng et al., 2024; Ye et al., 2024), Prompt Choreography is more general. Patterns resembling Prompt Cache (Gim et al., 2024) and Block-Attention (Sun et al., 2024), where encodings for a collection of static documents or prompts are precomputed, are also supported. Block-Attention integrates an extra position update step so that all retrieved messages have sequential positions, which is the default provided by our API when the offsets are omitted.

```
question = prefill(
    tokens='User: What is the capital of France?'
    parents=[]
)

docs = []
for doc in knowledge_base:
    docs.append(prefill(
        tokens=doc,
        parents=[]
    ))

resp = decode(
    header='Assistant:',
    parents=[*docs, question]
)
```

Rather than prefill and decode each of these messages, the API can be extended to express parallelism by overriding the functions to receive *lists* of choreography instructions and return *lists* of corresponding identifiers. We illustrate this, along with more complex caching patterns that can arise with these building blocks, in Figure 6.

3 Working with Modified Attention

3.1 A Baseline Approach

To contrast Prompt Choreography with the traditional Chat API (see §2.3), imagine a **naive implementation** of our API that mimics the Chat API, except for being invoked via two methods (`prefill`, `decode`) instead of one (`generate`).

The naive implementation does not cache any neural encodings. Each identifier now refers to just the *text* of a message, not its contextual encoding:

- `prefill` does not call the LLM. It ignores the `parents` argument. It simply stores `tokens` 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 `id`s) 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*.

Importantly, each naive decode encodes its prompt from scratch, so each message in its `parents` will attend (only) to all previous messages in `parents`.[6]

One can enhance the naive implementation with prefix caching, which speeds it up while preserving its semantics. We call this the **baseline method**.

Prompt Choreography differs because when `prefill` or `decode` creates a new input or output message, respectively, it also computes and stores a

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

contextual encoding of that message. These cached contextual encodings are reused whenever a message is reused—even if the message appears in a new context! 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 below.

Remark: Dohan et al. (2022) characterize a baseline workflow as a graphical model, where each random variable is a text string and depends on its parent variables. Input and output messages correspond to observed and unobserved variables. Prompt Choreography is the same, except that each random variable is now an *encoded* text string. This makes it faster to sample a variable given its parents, but changes the graphical model's semantics.

3.2 Distillation (via Fine-Tuning)

When information blockage or information leakage harms performance, we may attempt to recover baseline-level accuracy—without dropping the speedup—through lightweight 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 Llama 3.1 8B (Llama Team, 2024), decoding at temperature 0.7. For PEFT, we train LoRA adapters (Hu et al., 2021) with a fixed hyperparameter setting.[7] PEFT modifies $< 1\%$ of the model parameters while requiring 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. 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.

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

MultiQA We 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 says "Answer all questions." We compare three approaches (Figure 4): the **baseline** workflow allows question #2 to attend to question #1, the **choreographed serial** workflow drops this cross-attention but still offsets question #2 after question #1, and the **choreographed parallel** workflow completely eliminates question order by placing both questions at the same offset 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. As Table 1 shows, in the serial case it prefers to answer the *second* question (61.0% 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.

We then apply our fine-tuning recipe over 200 examples for 2 epochs to **choreographed parallel** and evaluate on 500 held-out question pairs. Fine-tuning demonstrates strong improvement over the untrained choreographed implementation, recovering most of the baseline performance in each column. Potentially the gap could be closed further with more examples or more epochs.

Branch-Solve-Merge To assess blockage in a more realistic setting, we implement a BSM workflow for the CommonGen task (Lin et al., 2020), which requires generating a coherent story incorporating a set of 30 concepts.[8] The workflow involves:

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

[8] We use training, development, and evaluation splits of size 100, 50, and 50, respectively.

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).

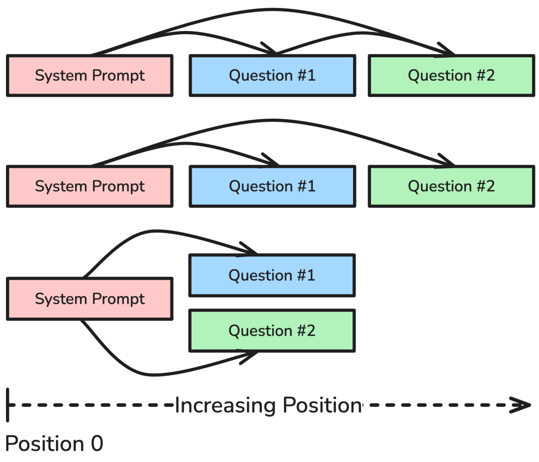


Figure 4: Attention topologies analyzed in MultiQA. Boxes represent messages, solid arrows represent attention dependencies, and horizontal displacement represents relative position, starting from position 0 on the left and increasing rightwards. **Top** depicts baseline, **middle** depicts choreographed serial, and **bottom** depicts choreographed parallel.

(1) a **branch** step that divides concepts into two 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. The two sub-stories are analogous to the two questions in MultiQA, so we compare the same workflows for prefilling them and using them as parents.

Both choreographed workflows perform substantially worse than the baseline when using the un-tuned 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 stories for 4 epochs, makes it statistically indistinguishable from the baseline.[9] Head-to-head comparisons judged by an LLM indicate that fine-tuning restores narrative quality, not merely coverage. Meanwhile, the baseline's unwanted positional bias is—of course—eliminated by our use of a parallel workflow.

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, information 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* information leakage, we use a workflow based on the classic Prisoner's Dilemma. Two game-playing agents, Alice and Bob, each privately develop a strategy through a chain of thought. They then engage in two rounds of open conversation before each 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). Llama 3.1 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 strategic thoughts. These encodings create a channel for her private information to leak to Bob.[11]

[9]It even does slightly better on our particular test set of size 50, but the confidence interval on the population difference is somewhat wide. A larger test set would be desirable.

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

[11]This can be avoided if Bob only sees versions of Alice's utterances that have been *freshly re-encoded* without this attention, as in the standard Chat API. Our API can recreate this

	Concept Coverage (%)			*Winrate vs. Baseline (%)*	
Implementation	**Average (Diff. CI)**	**Story 1**	**Story 2**	**Baseline Wins (CI)**	**Choreo. Wins (CI)**
Baseline	**81.0**	**87.6**	**82.4**	—	—
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:** Concept coverage metrics showing the percentage of concepts successfully incorporated into the final story. Group 1 and Group 2 denote the percent of concepts incorporated from each of two sub-stories generated in the solve step. We report 95% confidence intervals on the difference from baseline, obtained via the paired bootstrap. Underlining denotes a statistically significant differences in Group 1 and Group 2 coverage ($p < 0.05$, obtained via bootstrapping). Bolding denotes best performance or not significantly worse ($p > 0.05$, obtained via bootstrapping) **Right:** Head-to-head win-rates as judged by GPT-4o (remaining percentage represents ties), with 95% CIs obtained via bootstrapping.

	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 (Alice utility, 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 Exact Test.[10]

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, we improve statistical power by constructing paired examples. To construct a pair, first we run the baseline workflow. Then we run the choreographed workflow with the same system prompts, but prefill the strategic thoughts by copying them from the baseline workflow, rather than decoding new ones. We allow the subsequent conversations and decisions to diverge from the baseline workflow. We use fixed random seeds to generate separate 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 remainder.

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% → 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? Bob may be primed by the fact that these messages *mention* defecting. As an ablation, we generate games where Bob attends to encodings that are re-encoded to *only* attend to Alice's system prompt *or* her private plan (Figure 8). The results do weakly suggest that Bob is sensitive to the content of the leaked prompts. 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 conduct an additional experiment where Bob is explicitly prompted to predict Alice's decision after the conversation phase, to determine if Bob

by calling `prefill` on textual copies of Alice's utterances.

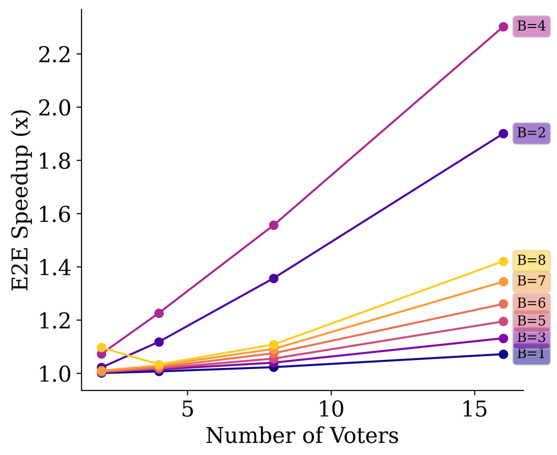


Figure 5: E2E speedup across varying numbers of branches and voters. In the ToT workflow, "branches" are candidate solutions generated for an problem, while "voters" denote independent agents that vote on the most promising solutions. Interestingly, the speedup is non-monotonic with respect to the number of branches and peaks for $B = 2$ and 4. This reflects a non-linear relationship: while re-encoding a large number of branches is expensive, it also introduces overhead to downstream decode calls that need to attend the longer context.

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 Exact Test.

Workflow	TTFT Ratio (CI)	E2E Ratio (CI)
ToT	3.5 $(3.26, 3.82)$	1.031 $(1.026, 1.036)$
MADiter	2.0 $(1.94, 2.07)$	1.036 $(1.028, 1.045)$
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.

actually discerns Alice's strategy. However, Bob's prediction accuracy revealed no statistically significant differences in any settings across 100 games (Table 6). Regardless, information leakage's impact on cooperation rate is notable. Crucially, fine-tuning effectively remedies this, leaving no statistically significant difference from the baseline.

4 Main Experiments

To evaluate Prompt Choreography across diverse real settings, we implement three workflows representing common architectural patterns (Figure 2). 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 between agents over a shared conversation history.
2. **Tree of Thoughts (ToT)** (Yao et al., 2023), with *hierarchical* exploration of multiple candidate solutions in parallel, followed by voting steps to extend the most promising solutions.
3. **Parallel Multi-Agent Debate (MADpar)** (Du et al., 2023), showcases a *strongly-connected* communication topology, where many agents generate in parallel while conditioning on the outputs of other agents from previous rounds.

4.1 Task Accuracy

We apply the fine-tuning recipe to Llama 3.1 8B detailed in §3.[12] 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,[13] 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 very fast **direct** workflow that prompts the LLM to answer the problem in a single step. Finally, we distill each full workflow: we try to fine-tune the LLM so that the fast direct

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

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

workflow produces the same final solution that the baseline method obtains using the full workflow.[14]

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 poor distilled performance indicates that fine-tuning by itself is not sufficient: the (choreographed) workflow at inference time is contributing something.

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 key metrics: (1) average time-to-first-token and (2) end-to-end wall-clock time (E2E). The former 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 (6.2x 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 (2.0x 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, around 1.03x (Table 5). This is expected, as workflow run-time is commonly dominated by decoding, which amortizes redundant computation. Regardless, a 3% E2E improvement can still be valuable and accumulate into significant long-term savings.

Examining Appendix A.3, we see that the E2E speedup for ToT and MADpar is *positively* correlated with the number of tokens generated in the workflow, indicating that they are more *prefill-bound*, whereas MADiter has a negative correlation. This is reasonable, as ToT and MADpar may become bottlenecked by re-encoding large contexts for all agents in parallel at certain steps. We run ToT across a broader range of configurations, and indeed find that prefilling begins to dominate in the baseline as we scale critical parameters. This results in more dramatic E2E speedups (>2.2x, §4).

[14]Either the final message or, for MADiter and MADpar, an answer extracted programmatically from the final message.

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 allows selective reuse of *static* prompt components, neither handles reusing content generated at run-time or arbitrary context reordering. Some methods specialize caching to information retrieval (Sun et al., 2024; Lu et al., 2024; Wang et al., 2025) by pre-computing knowledge bases and sometimes incorporating repositioning. Prompt Choreography again generalizes these approaches. In addition, prior work on efficient LLM inference, such as KV cache compression (Li et al., 2024) and LLM serving systems (Kwon et al., 2023; Zheng et al., 2024), are generally complementary to Prompt Choreography. While these optimize the low-level *components* of LLM inference, we show that the high-level *structure* of a workflow can also be optimized for better cache reuse.

6 Conclusions and Future Work

In this work, we introduce Prompt Choreography, a general framework for tackling redundant computation in LLM workflows through a dynamically managed, global KV cache. Although evaluated here on a single model and node, future work includes verifying generalization across more diverse models and scales, as well as deeper mechanistic analysis into how cached encodings impact information flow in workflows. On top of releasing our reference implementation, we also plan to extend these principles into production-ready serving systems, potentially integrating with existing widely-adopted systems such as vLLM.

Acknowledgements

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

References

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness.

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.

Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. 2024. Flex attention: A programming model for generating optimized attention kernels.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. 2023. Improving factuality and reasoning in language models through multiagent debate.

In Gim, Guojun Chen, Seung seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt Cache: Modular attention reuse for low-latency inference.

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.

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.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models.

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.

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 E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention.

Haoyang Li, Yiming Li, Anxin Tian, Tianhao Tang, Zhanchao Xu, Xuejia Chen, Nicole Hu, Wei Dong, Qing Li, and Lei Chen. 2024. A survey on large language model acceleration based on kv cache management. *arXiv preprint 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.

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.

Llama Team. 2024. The llama 3 herd of models.

Songshuo Lu, Hua Wang, Yutian Rong, Zhi Chen, and Yaohua Tang. 2024. Turborag: Accelerating retrieval-augmented generation with precomputed kv caches for chunked text.

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

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative agents: Interactive simulacra of human behavior.

Ofir Press, Noah A. Smith, and Mike Lewis. 2022. Train short, test long: Attention with linear biases enables input length extrapolation.

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, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. 2023. Roformer: Enhanced transformer with rotary position embedding.

East Sun, Yan Wang, and Lan Tian. 2024. Block-attention for efficient rag.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need.

Xi Wang, Taketomo Isazawa, Liana Mikaelyan, and James Hensman. 2025. Kblam: Knowledge base augmented language model.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models.

Lu Ye, Ze Tao, Yong Huang, and Yang Li. 2024. Chunkattention: Efficient self-attention with prefix-aware kv cache and two-phase partition.

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.

Appendices

A Workflow Details

For each workflow, we follow the prompts provided by the original papers and reference implementations. Further details can be found in Liang et al. (2024) (MADiter), Yao et al. (2023) (ToT), and Du et al. (2023) (MADpar). Evaluation is done at temperature 0.7. Pseudocode is provided in Figure 6.

A.1 Iterative Multi-agent Debate

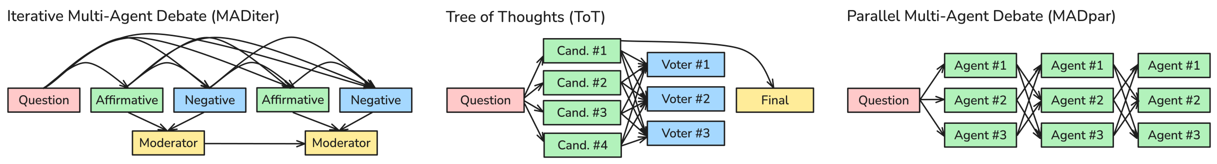
Three maximum rounds of debate with moderator-led early-stopping and level 2 (default) "debate-level" prompt.

A.2 Tree of Thoughts

One-level breadth-first search, with 8 solution branches followed by 4 votes. The best candidate is then expanded into the final solution.

A.3 Parallel Multi-agent Debate

Three agents over three rounds of debate. We do not include the intermediate summarization step proposed by the original paper.



```
# prefill initial prompts
prompts = {...}

context = []
for _ in range(ROUNDS):
    # for each participant
    for role, header in [
        ('aff', 'Affirmative:'),
        ('neg', 'Negative:'),
        ('mod', 'Moderator:')
    ]:
        # decode new message
        msg = decode(
            header=header,
            parents=[
                prompts[role],
                *context
            ]
        )

        if role != 'mod':
            context.append(msg)

        # early stop?
        elif parse_stop(msg):
            break
```

```
# prefill initial prompts
prompts = {...}

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

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

# final answer from best cot
final = decode(
    header='Assistant:',
    parents=[
        prompts['gen_answer'],
        cots[parse_best(votes)]
    ]
)
```

```
# prefill initial prompts
prompts = {...}

prev = []
for _ in range(ROUNDS):
    tasks = []
    for i in range(AGENTS):
        # previous round answers
        others = [
            p for j, p in
            enumerate(prev)
            if i != j
        ]

        # create new task
        tasks.append({
            'header': f'Agent {i}',
            'parents': [
                prompt['question'],
                prompt['answer']
            ] + others
        })

    # decode all new answers
    prev = decode(update_tasks)
```

Figure 6: Pseudocode with accompanying figure for each workflow we analyze in §4.

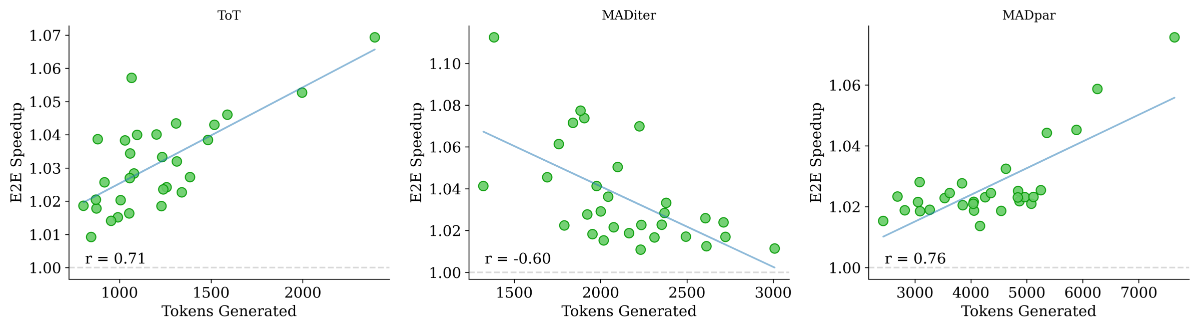


Figure 7: E2E speedup against total tokens generated for each workflow ($n = 30$).

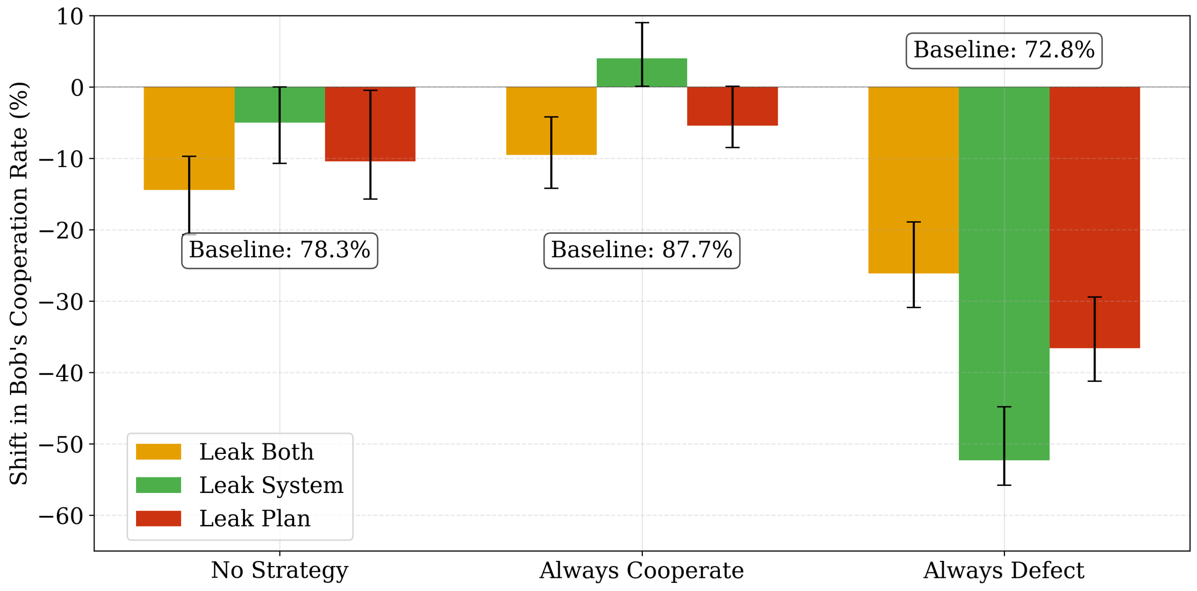


Figure 8: Shift in cooperation rates between choreographed and baseline implementations of the Prisoner's Dilemma. "Leak Both" is the normal choreographed implementation, whereas "Leak System" and "Leak Plan" are ablations that explicitly re-encode Alice's output encodings to strictly attend to her private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

Baseline					
Strategy	**Alice Actual**	**Bob Predicted**		***Outcome***	
	Cooperate	**Cooperate**	**Correct**	**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					
Strategy	**Alice Actual**	**Bob Predicted**		***Outcome***	
	Cooperate	**Cooperate**	**Correct**	**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					
Strategy	**Alice Actual**	**Bob Predicted**		***Outcome***	
	Cooperate	**Cooperate**	**Correct**	**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 6: Results from playing 100 games and prompting Bob to explicitly predict Alice's decision. 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.