MLSys

AgenticCache: Cache-Driven Asynchronous Planning for Embodied AI Agents

Hojoon Kim[1*], Yuheng Wu[2], Thierry Tambe[2]

[1]Seoul National University [2]Stanford University

*Work done during internship at Stanford

Stanford | ENGINEERING
Electrical Engineering

9th Conference on Machine Learning and Systems (MLSys 2026), Bellevue, USA

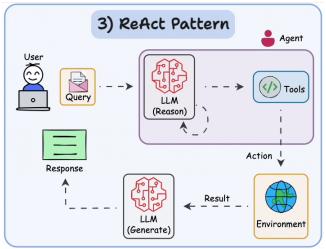
Table of contents

Agents are going real-world

They plan. They perceive. They act. — and before every move, they wait.

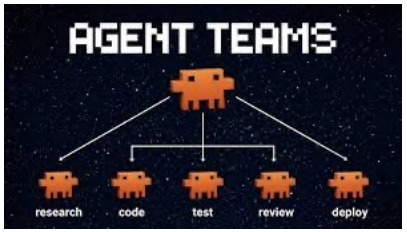
LLM agents

ReAct, Reflexion



Multi-agent

Coding Agent
Cooperative planning



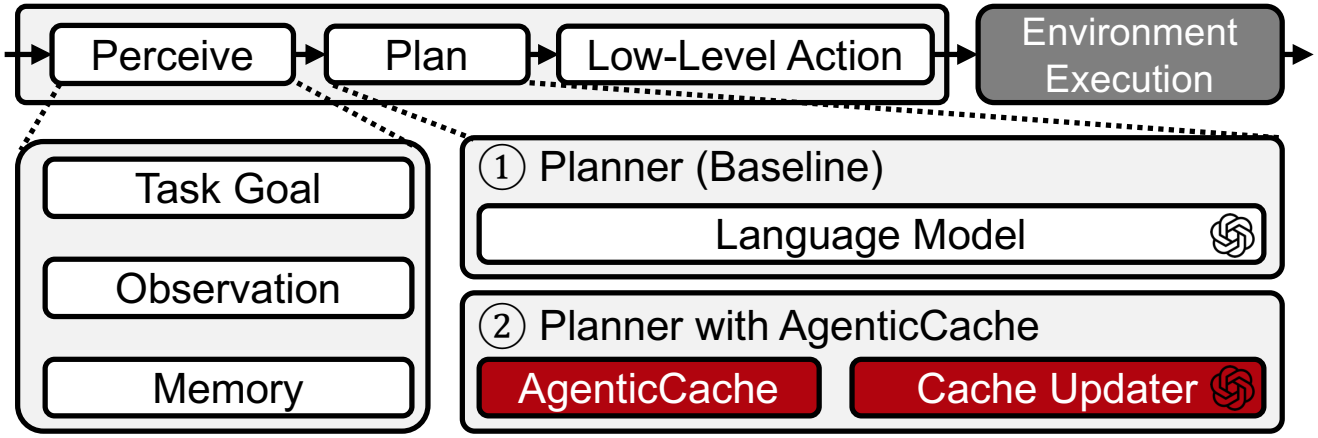
Embodied AI

ThreeDWorld,
BEHAVIOR

H. Kim et al., " AgenticCache: Cache-Driven Asynchronous Planning for Embodied AI Agents"

Background: an embodied agent's plan-act loop

Perception → Plan → Action, repeated until the task ends.



The Plan box is an LLM. Every iteration blocks on it.

Every step waits for an LLM

The plan–act loop is synchronous: act cannot start until plan finishes.

Latency	Tokens	Cost
Hours / episode	**Millions / run**	**$10s–$100s / run**

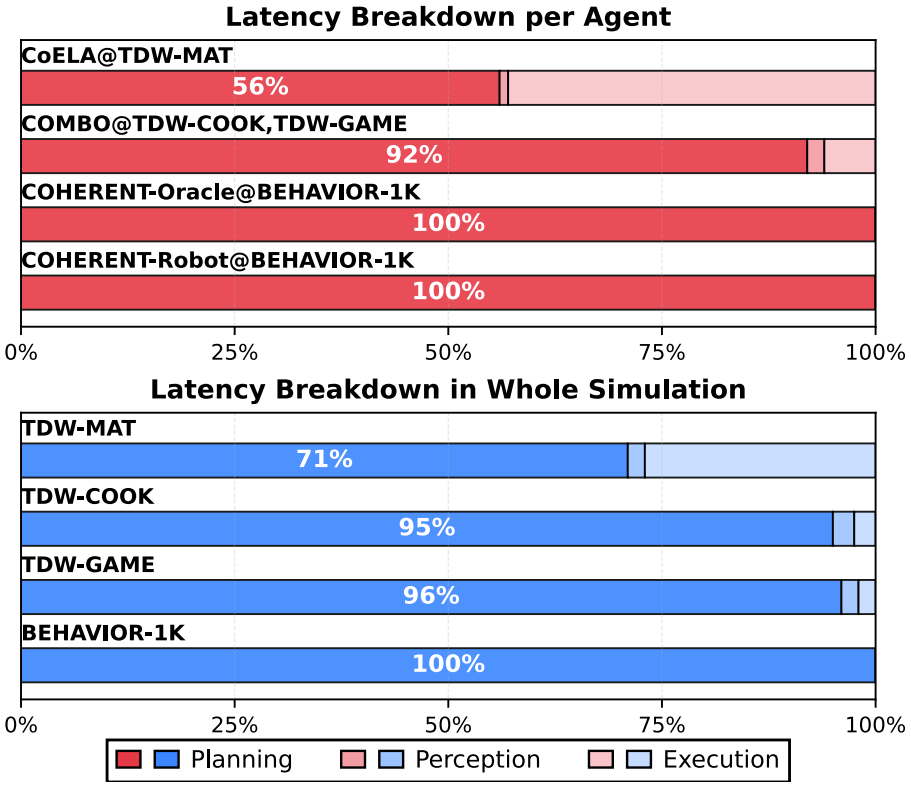
Every step waits for an LLM

The plan–act loop is synchronous: act cannot start until plan finishes.

Goal: take the LLM off the critical path.

Latency	Tokens	Cost
Hours / episode	Millions / run	$10s–$100s / run

The LLM eats the clock



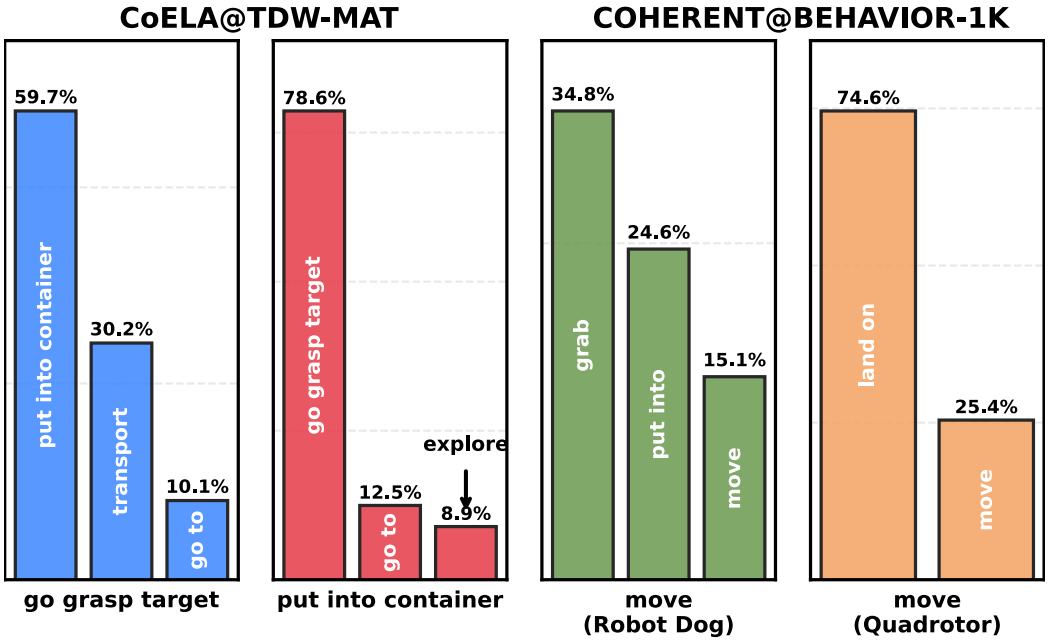
LLM planning is 70%+ of the runtime

Prior caches are not plan caches

LLM systems cache compute; embodied agents need to cache decisions.

Scheme	Granularity	What it reuses	Limitation
KV cache	Token-level	Reuse attention key/value tensors	Agent still invokes LLM per step
Context cache	Prompt-level	Reuse prompt-response pairs	Needs exact / close match
Template cache	Query-level	Reuse structured Query templates	Still invokes LLM per step
AgenticCache	**Plan-transition**	**Reuse 2-gram plan patterns + async LLM**	**Adapts online via updater**

But plans aren’t random



Plan locality

Sound familiar?

Computer architects faced the same problem — 40 years ago.

CPU branch predictor		Agents?	
History	Past branch outcomes	History	?
Predict	Taken vs not-taken	Predict	?
Speculate	Execute ahead of resolution	Speculate	?
Correct	Flush pipeline on miss	Correct	?

Our question

Can we build a branch predictor for agents?

Our answer

AgenticCache

Predict. Act. Correct — asynchronously.

Design goals

1. **Cut the critical path**
   Don't let planning block action.
   LLM off the loop

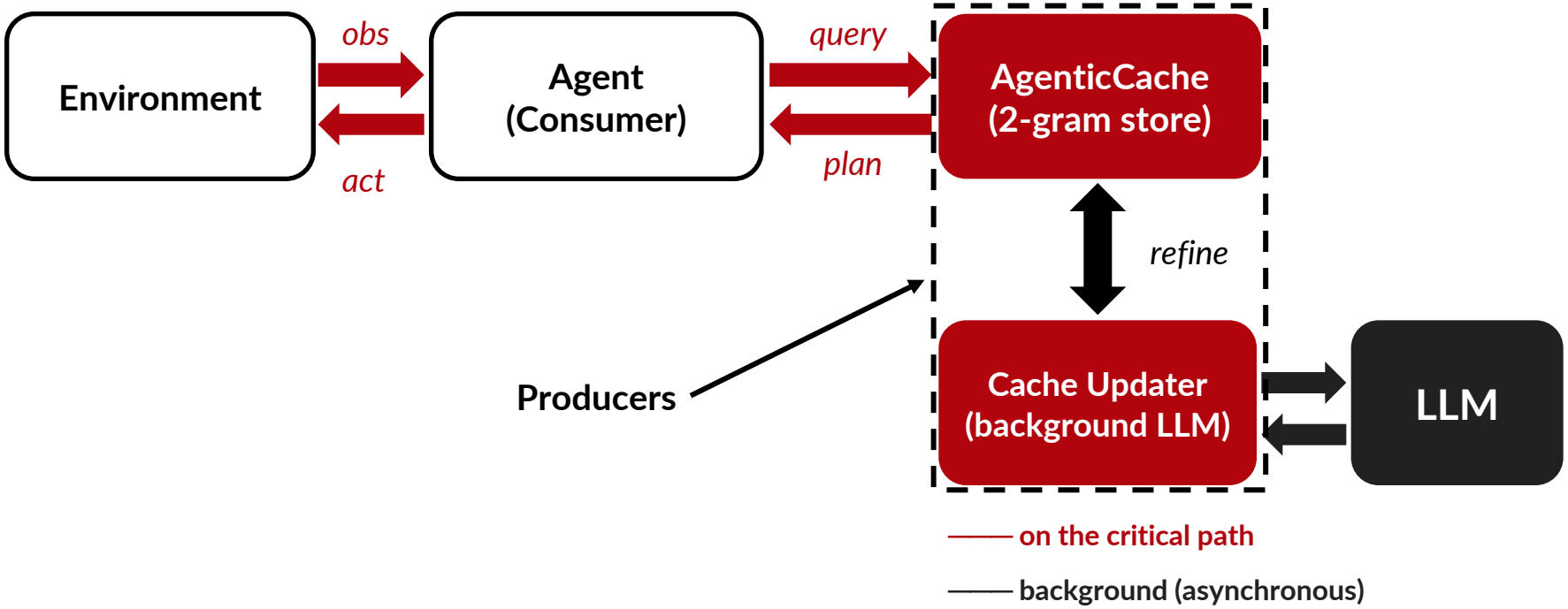
2. **Stay context-aware**
   A static cache fails; a self-correcting one adapts.
   Updater fixes stale plans

3. **Be cheap to deploy**
   Tiny cache, no extra training.
   Low overhead

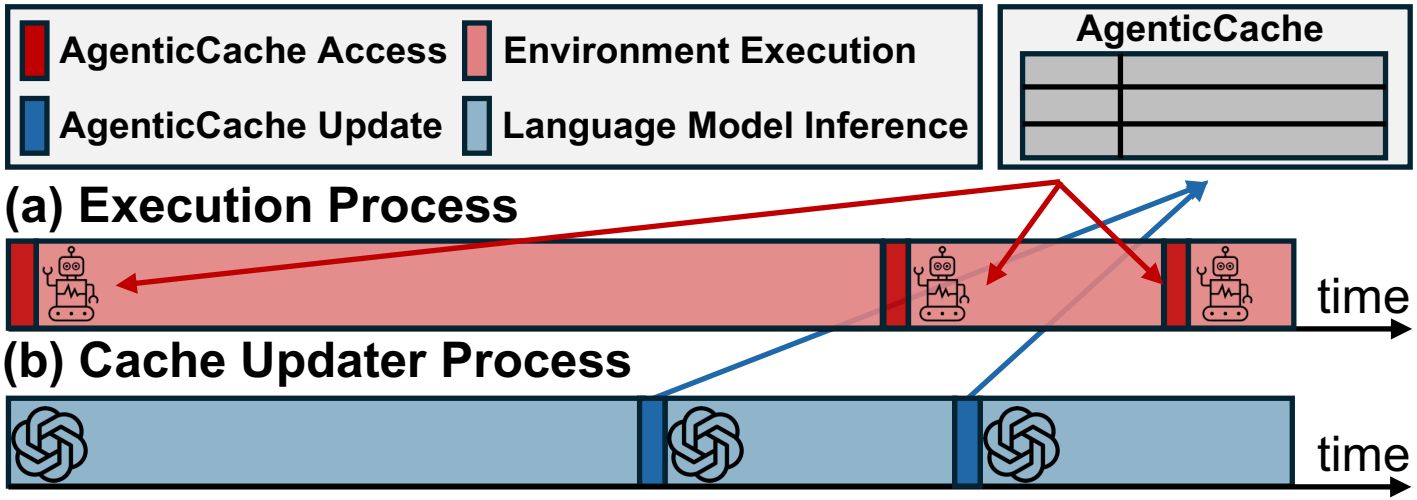
4. **Preserve task quality**
   Efficiency must not cost success rate.
   No SR regression

AgenticCache: system overview

Cache on the critical path. LLM moved to a background updater.



AgenticCache: runtime overview



Cache structure: 2-gram entries with state metadata

AgenticCache (2-gram store)

Cache Updater (background LLM)

Each entry stores a plan transition and the state range in which it's valid.

Prev	Next	Steps	#Items	#Fin.	#Rooms	C	I	Score
GoTo	Explore	[62,2970]	[0,3]	[0,10]	[0,5]	58	0.25	14.5
GoTo	**GoGrasp**	**[19,2645]**	**[0,3]**	**[0,9]**	**[1,4]**	**35**	**0.55**	**19.3**
GoGrasp	Transport	[22,2974]	[1,4]	[0,8]	[0,5]	60	0.17	10.2
GoGrasp	GoTo	[117,2410]	[0,4]	[0,6]	[0,5]	46	0.23	10.6

KEY (lookup) **VALUE (transition + stats)**

Why metadata ranges?

Entries fire only when the agent's current state falls inside every observed range.

Why 2-gram?

Long enough for locality; short enough to stay tiny.

Which cached plan do we pick? — Score = C × I

AgenticCache (2-gram store)

Cache Updater (background LLM)

Among entries whose metadata ranges admit the current state.

$$S(P_i \rightarrow P_j) = C(P_i \rightarrow P_j) \times I(P_j)$$

$C(P_i \rightarrow P_j)$ — local signal: per-pair transition history (how often P_j follows P_i)

$I(P_j)$ — global signal: overall importance of P_j across the cache

The Cache Updater: a background LLM process

AgenticCache (2-gram store)

Cache Updater (background LLM)

While the agent acts, the updater asks the LLM what it would have chosen.

✓ Confirm — LLM agrees with the executed plan → reinforce.

✗ Correct — LLM disagrees → add / update transition, swap the ongoing.

Results

Averaged across 4 benchmarks × 3 models (12 configurations).

−65% latency

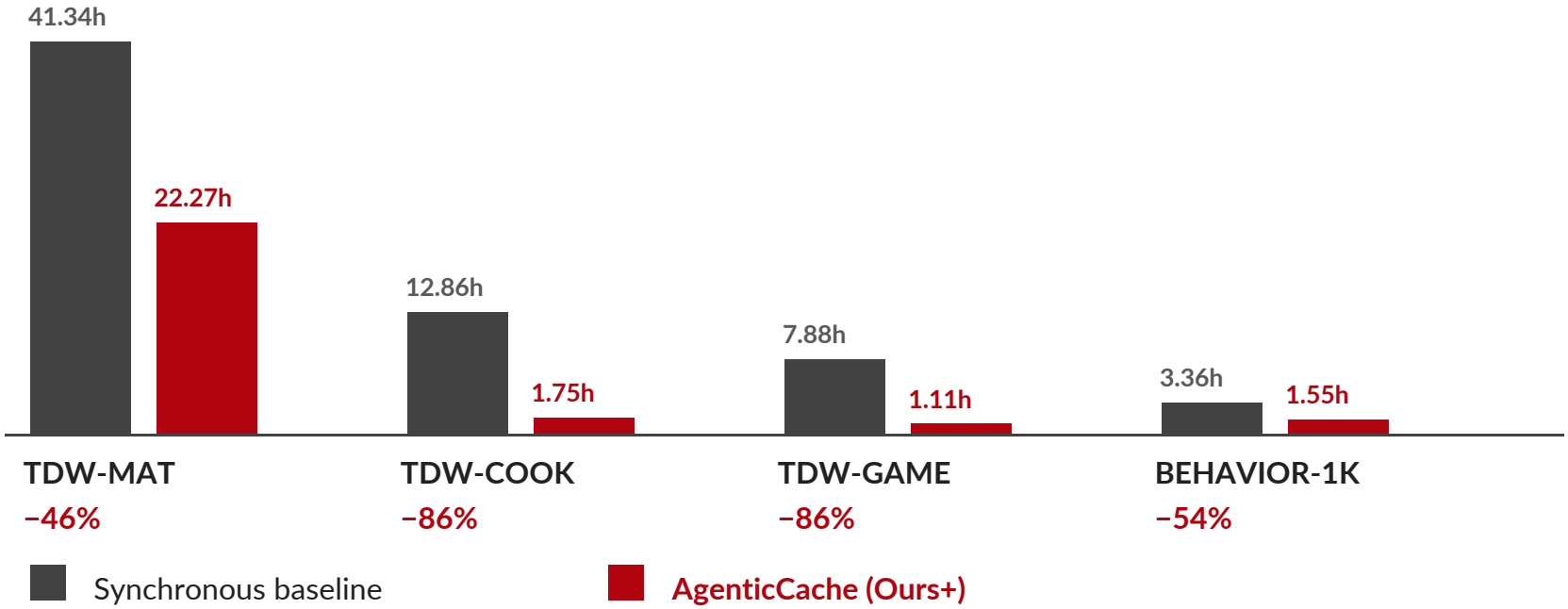
−50% tokens

+22% success rate

Efficiency without sacrificing task success.

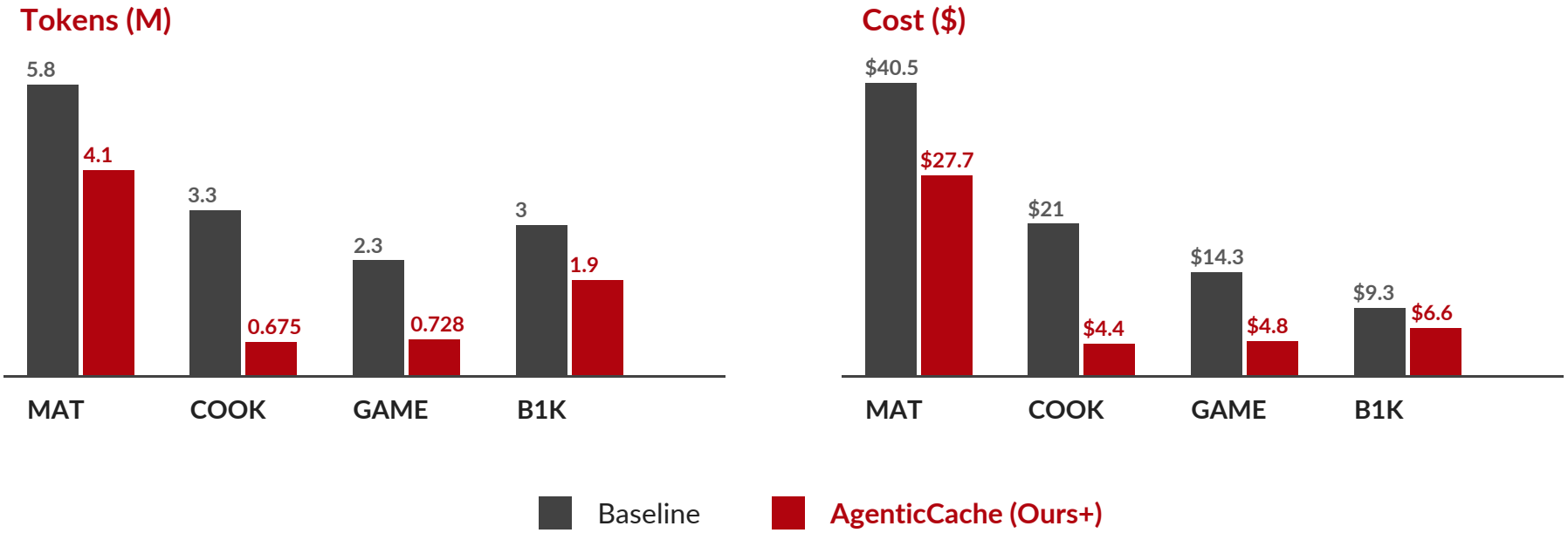
Latency: hours saved, per benchmark

GPT-5, end-to-end episode latency (hours). Lower is better.



Tokens & cost: half the bill

Token usage and $ cost per full evaluation (GPT-5).



Suppression kills redundant LLM queries — the headline savings.

Limitations & Future Work

Limitations

- Multi-agent contention: cache is per-agent; shared resources need coordination.
- Domain scope: evaluated on structured tasks; open-ended domains may weaken plan locality.

Future Work

- Higher-order/hierarchical transitions, priority-based coordination, adaptive cache policies.
- Broader AI agents: beyond embodied — multi-agent teams, coding agents.

Takeaways

1

Plan locality

Agents repeat the same short plan transitions. Don't recompute them.

2

Decouple planning from acting

Put the LLM in the background. The cache is the fast, on-path planner.

3

Cache + updater = branch predictor for agents

Predict, speculate, correct — with order-of-magnitude latency / cost wins.

−65% latency · −50% tokens · +22% success rate

Thank you

AgenticCache

Cache-Driven Asynchronous Planning for Embodied AI Agents

Hojoon Kim · Yuheng Wu · Thierry Tambe

Seoul National University · Stanford University

Questions?

Paper:

arXiv (MLSys´26)

Code:

github.com/hojoonleokim/MLSys26_AgenticCache