Agentics Day: MCP + Agents
NORTH AMERICA

STOP WASTING GPUs

Workload-Aware Caching for Agentic AI Workflows on Kubernetes

Research • MCP architecture • Real Argo GPU execution



Chetan Sharma

Agentics Day: MCP + Agents · North America 2026

SPEAKER INTRODUCTION

About the Speaker



Chetan Sharma

Senior Software Engineer, Google

Specializes in large-scale data processing systems, including streaming and batch pipelines and data storage infrastructure.

TODAY'S FLOW

From Repeated Work to Verified GPU Avoidance

Five steps from the research idea to a practical Kubernetes deployment

01

The reuse problem

Where agent workflows repeat expensive intermediate work

02

Better retention

Dependency, cost and frequency signals; original research

03

Safe MCP integration

Verified identity, eligibility, single-flight and dispatch

04

Kubernetes results

Matched workload tests and actual GPU task-pod launches

05

Practical choices

When reuse pays off and how to measure avoided work

Key question: what real GPU work does each cache hit avoid?

THE PROBLEM

Why Agentic Workflows Repeat Expensive GPU Computation

SUPPLIER AUDIT REPORT · V1

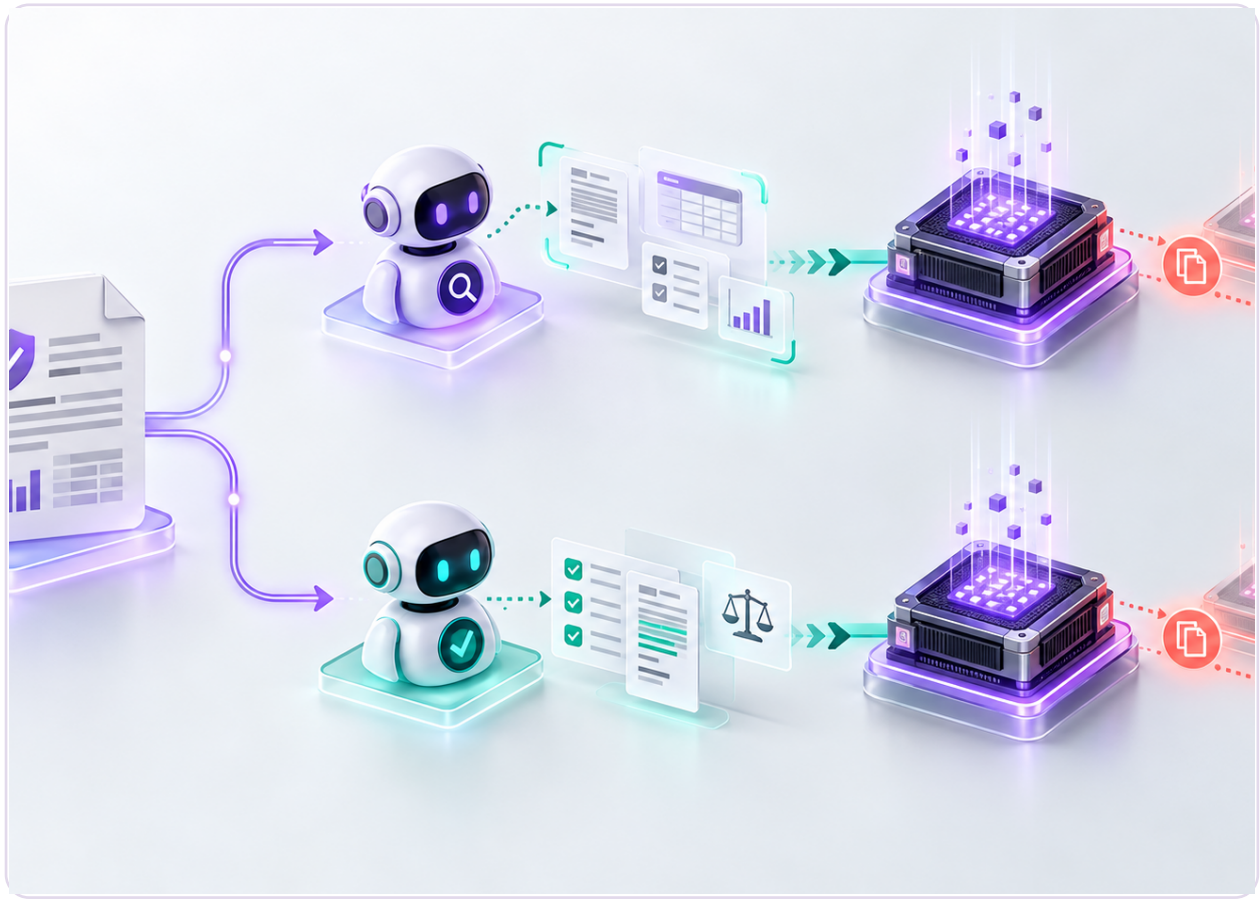
Sub-Agent 1: Analyst

Reads the report, generates embeddings, retrieves evidence, and produces a risk assessment.

Sub-Agent 2: Reviewer

Validates the findings and rechecks evidence—requiring the same document representation later.

Different agent responsibilities. Same intermediate computation.

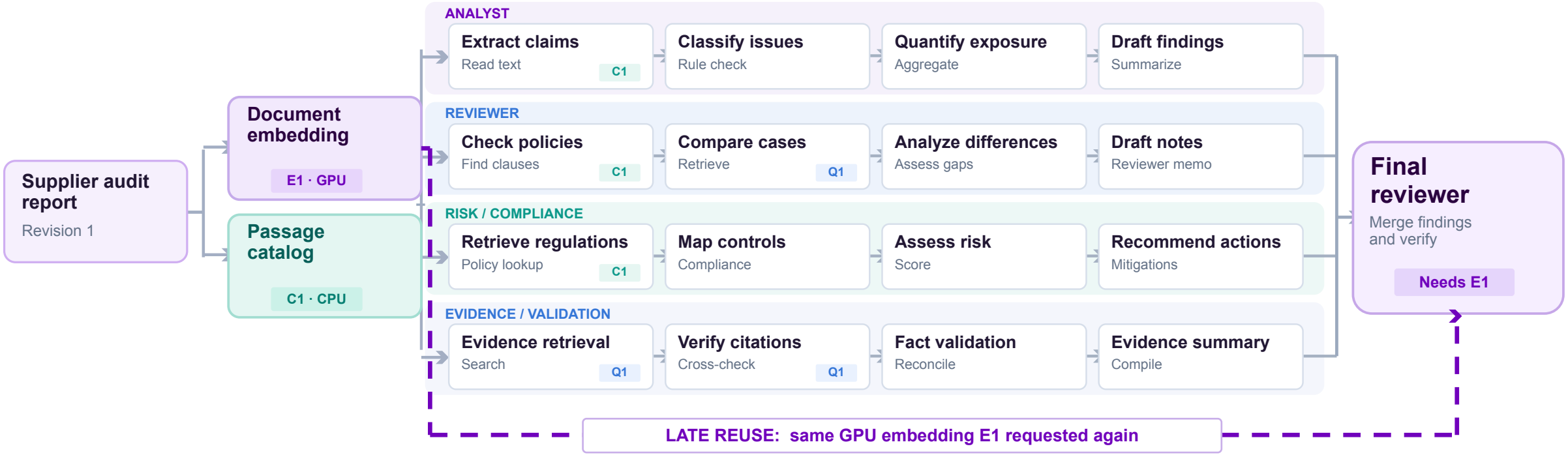


The waste is duplicate intermediate computation—not duplicate agent responsibilities.

THE PROBLEM

Complex agent workflows hide reuse opportunities

One supplier report → four parallel agent branches → downstream tasks → late reuse of the same GPU embedding



REUSABLE ARTIFACT IDS

E1	GPU embedding
C1	CPU passage catalog
Q1	Retrieved evidence (same query)

Matching IDs are reusable only when content, model, request identity, and sharing scope also match.

E1 may go unused through many tasks—then become valuable again at final review.

THE LIMITATION OF LRU

Cache Helps! But What Should We Evict?

1 CACHE IS FULL



Analyst has produced two reusable artifacts

CACHE · 2 / 2

A Report embeddings · older

B Risk features · recent

2 NEW RESULT ARRIVES



Artifact C needs space

B

C

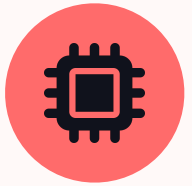
LRU evicts A

3 REVIEWER NEEDS A



Reviewer asks for Artifact A

CACHE MISS



GPU runs again

LRU knew what was used last. It did not know what the workflow would need next.

THE KEY INSIGHT

What if the Cache Knew What Comes Next?



Artifact A

Older in access history—but still feeds Reviewer + Compliance.

Artifact B

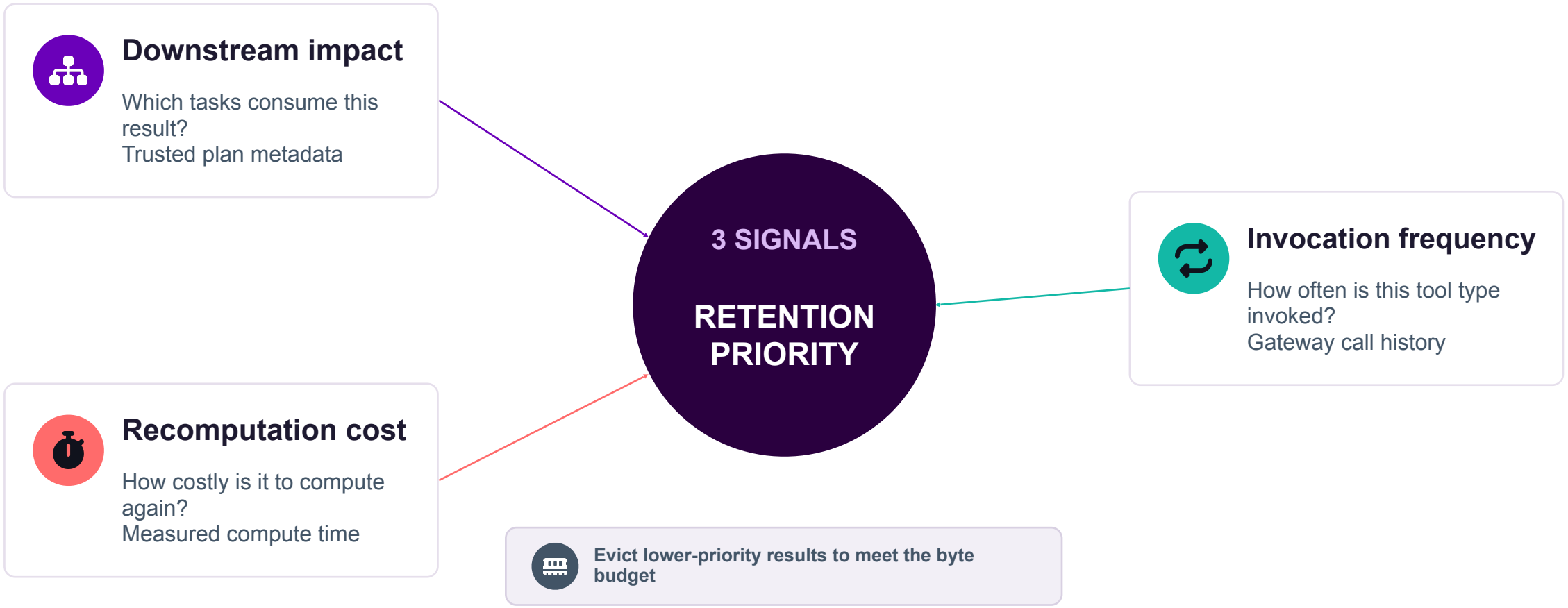
Used recently—but no later reuse is planned.

Keep A • Evict B

An older artifact may be more valuable than a recently accessed one.

RETENTION STRATEGY

What Makes an Artifact Worth Keeping?



Study policy: combine three signals into an insertion-time retention score.

ORIGINAL RESEARCH

The Research: Evaluating Workload-Aware Caching

Published benchmark scale • Plan-Act multi-agent DAGs • Fixed cache capacity within each benchmark

MP-DocVQA

MULTI-PAGE DOCUMENT QA

~5.9K
documents

- ~46.4K questions
- 1–20 pages / document
- 1–85 queries / document

Cache budget 500 MB

SlideVQA

PRESENTATION QA

2,619
slide decks

- ~14.5K questions
- 20 slides / deck
- 1–13 queries / deck

Cache budget 500 MB

VideoMME

VIDEO UNDERSTANDING

900
videos

- 2,700 questions · 254 hours
- 11 sec – 1 hr / video
- 3 queries / video

Cache budget 5,000 MB

WHAT WE COMPARED

Latency
Average query / pipeline time

Cache behavior
Hits and retention decisions

Answer quality
ANLS or video QA accuracy

FINITE-CACHE POLICIES

Workload-aware | LRU | LFU | FIFO | ARC

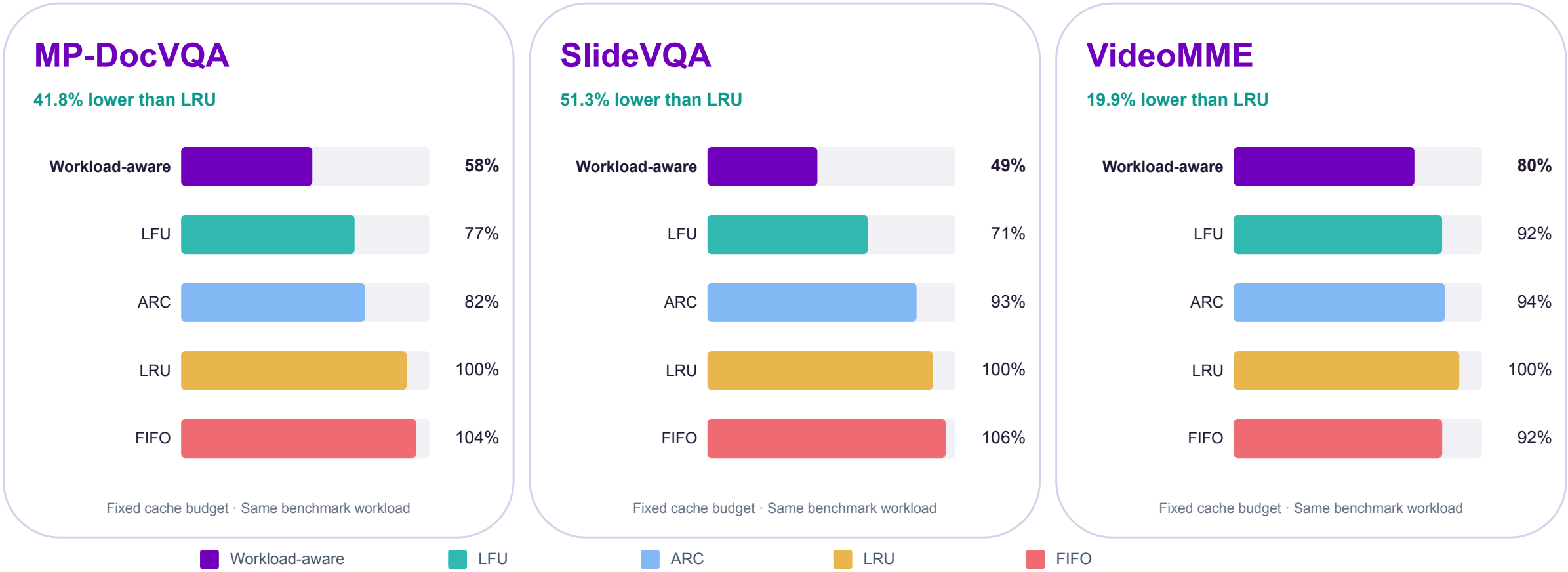
Different source types and reuse patterns stress the same retention decision.

Sources: arXiv:2607.20495

ORIGINAL RESEARCH RESULTS

Up to 51% Lower Pipeline Latency Than LRU

Average query latency normalized to LRU = 100% • Lower is better



Workload-aware had the lowest average latency among finite-capacity policies in all three benchmarks.

Source: Table I, Workload-Aware Caching for Multi-Agent Systems (arXiv:2607.20495)

WHY GPU SAVINGS MATTER

Can We Avoid GPU Work on Kubernetes?

GPU

Why avoiding GPU tasks matters

Accelerator capacity is higher-priced and often shared. Redundant tasks add scheduling pressure and compete with useful GPU work.

Avoid redundant tasks. Preserve accelerator capacity.

THE QUESTION WE TESTED

Can smarter caching prevent unnecessary GPU tasks from launching on Kubernetes?

TWO THINGS WE NEEDED TO VERIFY

CORRECTNESS

Equivalent results, valid revisions, and authorized reuse?

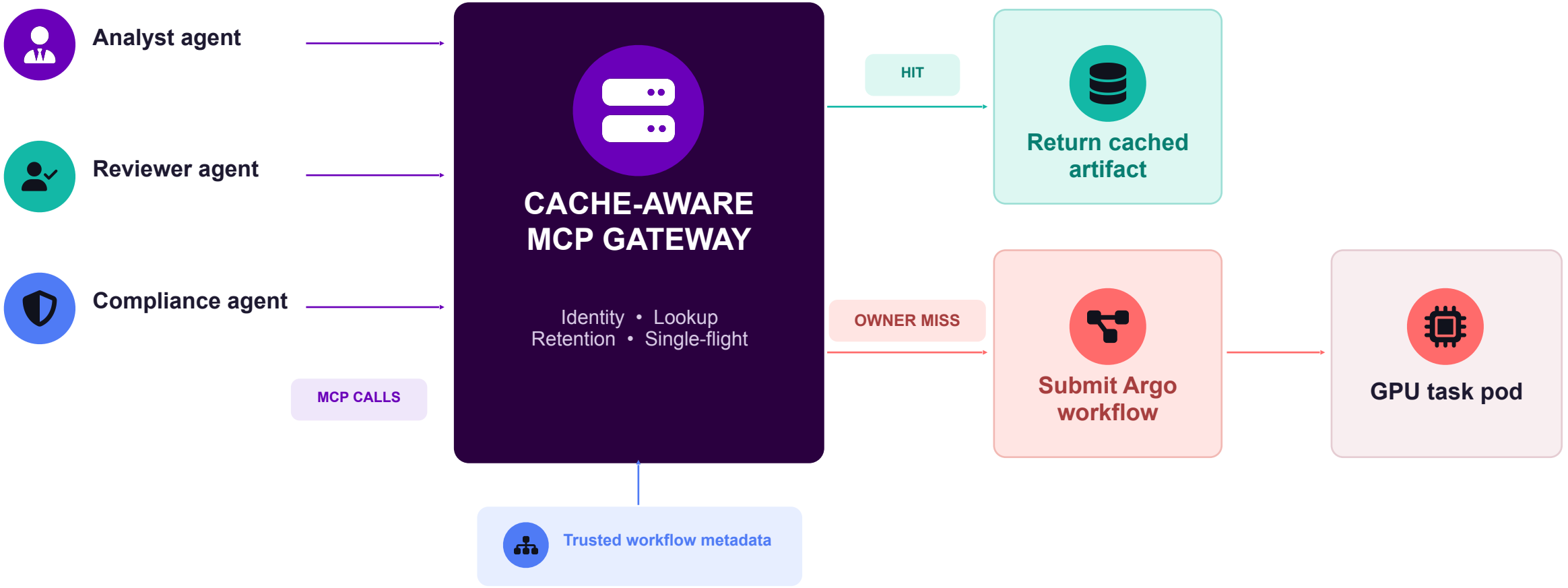
EFFECTIVENESS

Does workload-aware retention avoid more real GPU launches than standard LRU?

Goal: avoid unnecessary GPU execution while preserving correctness.

SYSTEM ARCHITECTURE

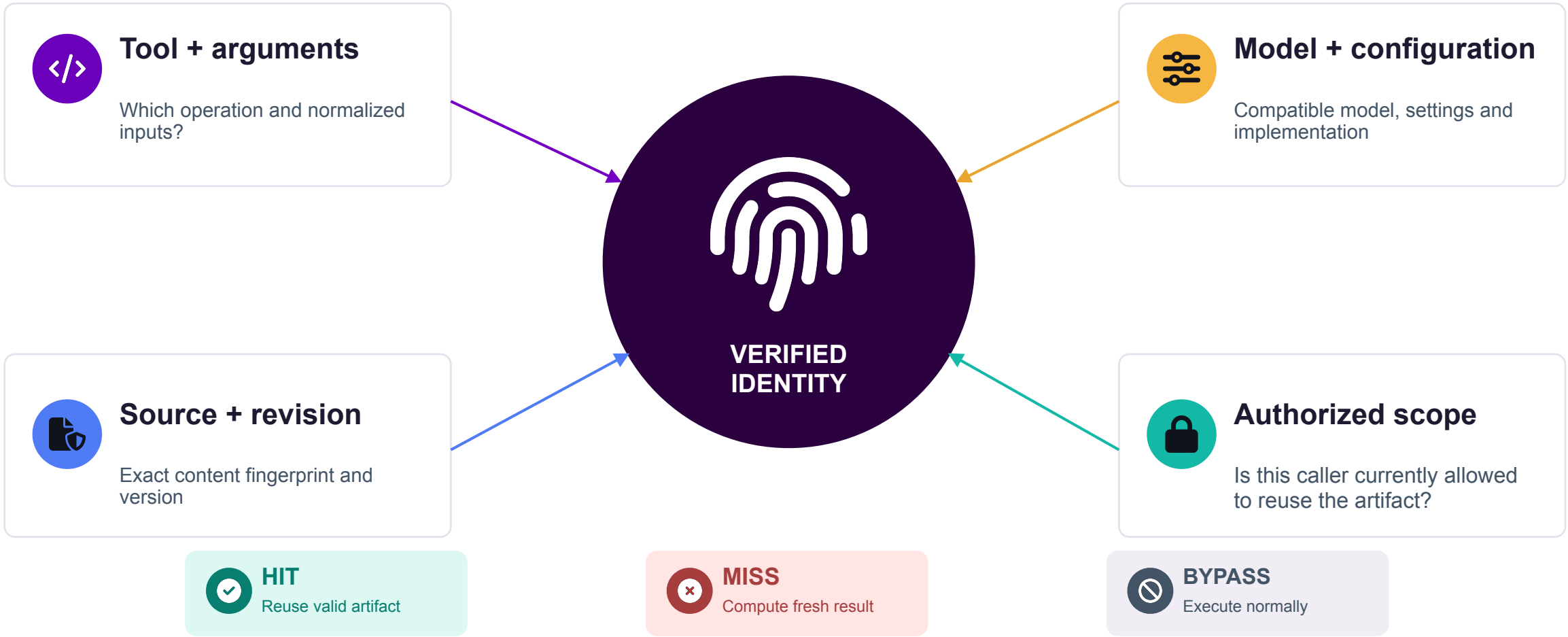
Cache Before GPU Dispatch



Reuse before dispatch can avoid both a new GPU task pod and recomputation.

CACHE CORRECTNESS

When Is It Safe to Reuse a Cached Result?



MISS = eligible but absent. BYPASS = excluded, with no cache lookup or store.

IMPLEMENTATION PLAYBOOK

Make Your MCP Workflow Cache-Aware in 4 Steps

Keep your existing agents and GPU workers. Add intelligence at the execution boundary.



01

Route MCP requests

Point selected tools at the cache-aware endpoint.



02

Identify cacheable tools

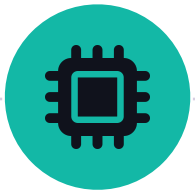
Allowlist safe tools; fingerprint source, model and scope.



03

Connect workflow metadata

Supply trusted DAG + artifact-consumer mapping.



04

Connect task execution

Dispatch owner misses to CPU or GPU executors; validate outputs.

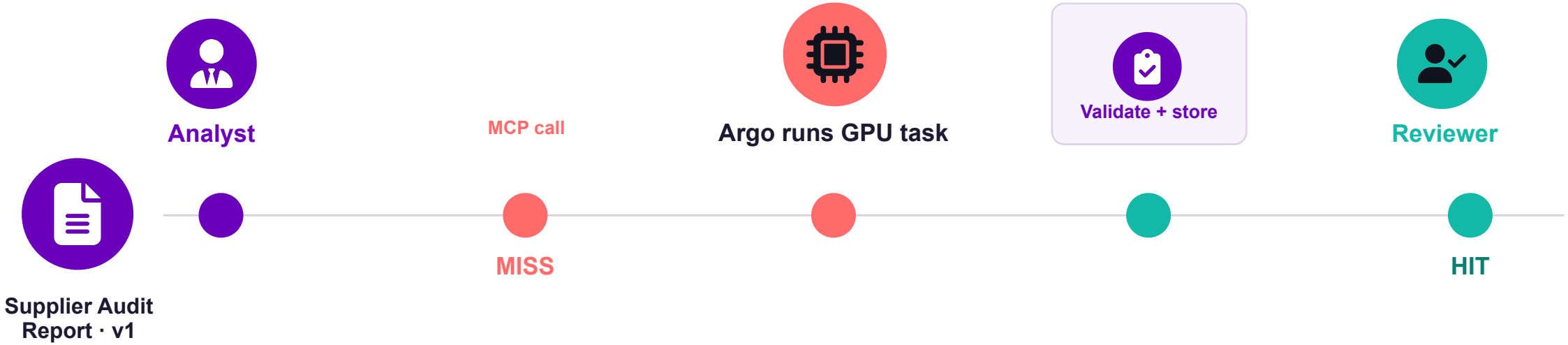
CACHEABLE · embed_document / catalog_passages

BYPASS · record_review / live_status

The pattern works with Argo — or another orchestrator, provided misses can be dispatched and observed.

REQUEST WALKTHROUGH

One Document. Two Agents. One GPU Execution.



FIRST REQUEST

No matching artifact exists. One GPU computation produces the reusable embedding.

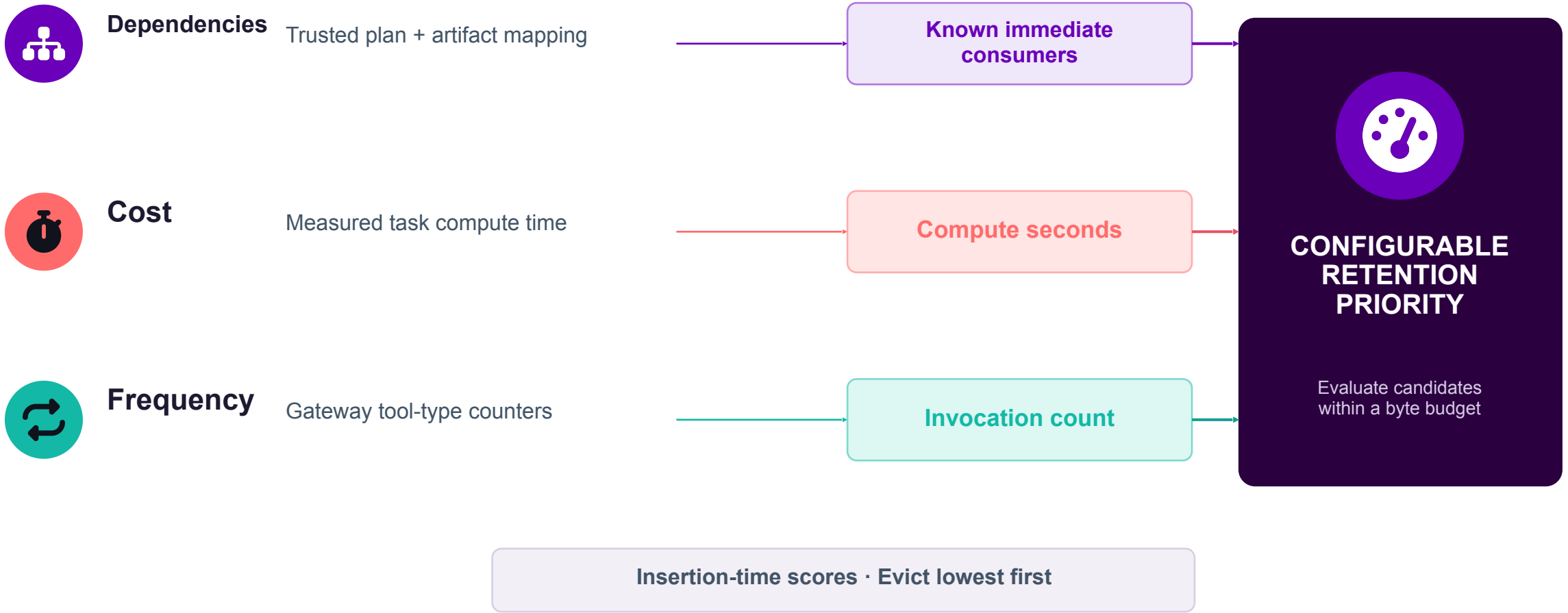
LATER REQUEST

If still retained, the matching result is reused.
The Reviewer performs its own validation.

Share the intermediate computation — not the agents' responsibilities.

SIGNAL COLLECTION AND EVICTION

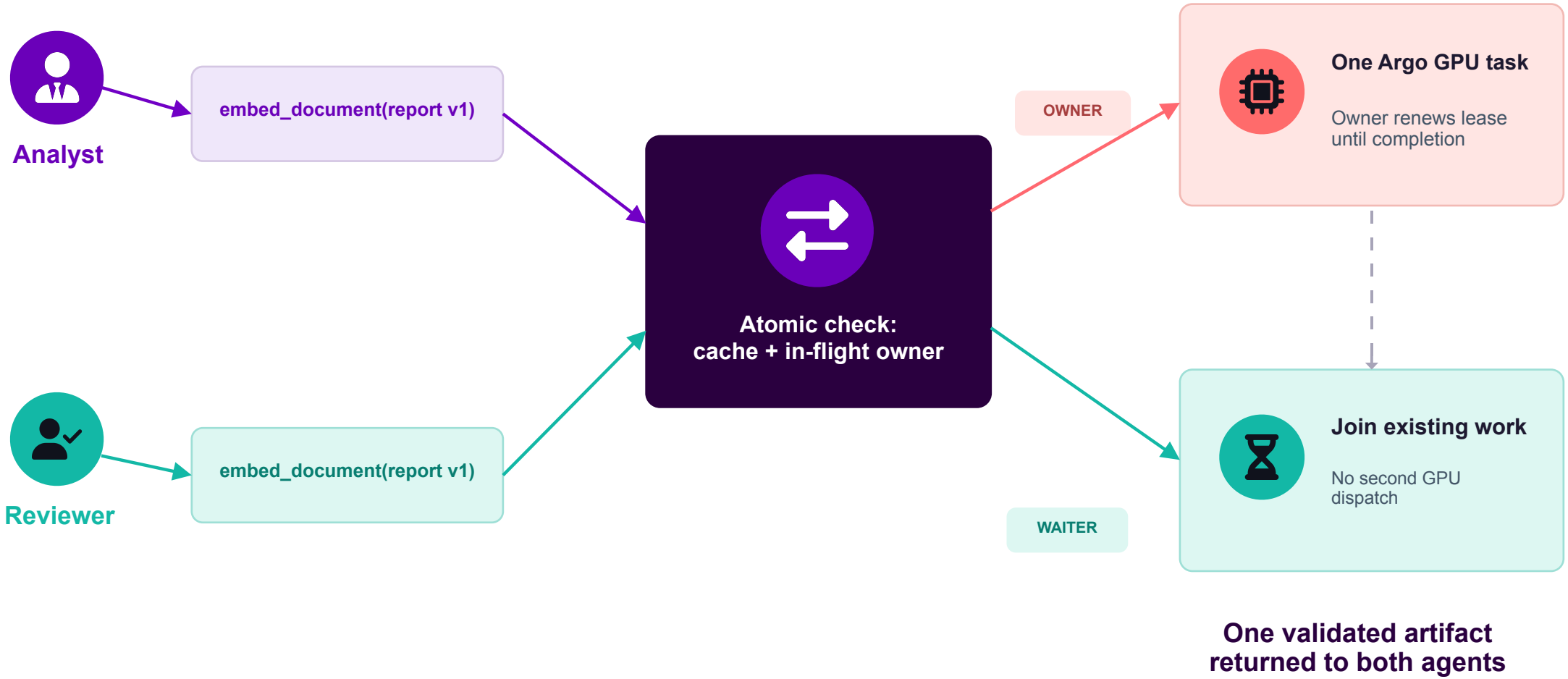
How Does the Cache Make Retention Decisions?



Use observable system signals, not the agent self-importance claims.

SINGLE-FLIGHT EXECUTION

What Happens When Multiple Agents Request the Same Artifact?

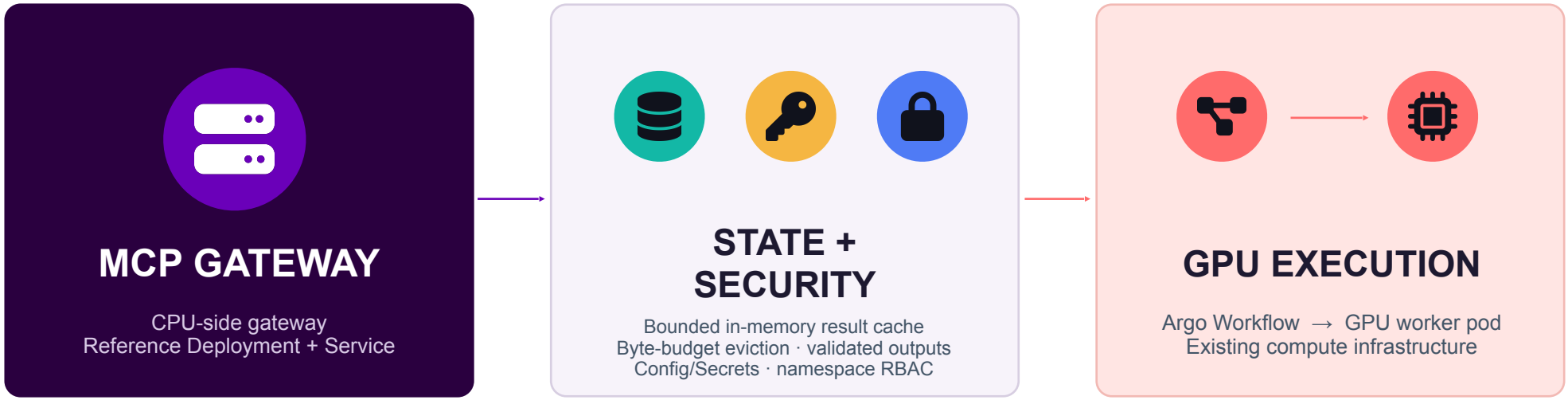


Two concurrent requests should not automatically create two GPU tasks.

KUBERNETES DEPLOYMENT

Deploying the Cache-Aware MCP Gateway on Kubernetes

KUBERNETES CLUSTER



Trusted workflow metadata connects DAG tasks to reusable artifacts

The gateway manages reuse. Argo manages execution.

CONTROLLED EXPERIMENT

How Did We Evaluate It on Kubernetes?

Same request trace • Same cache budget • Same model • Same GPU environment



High-fanout delayed reuse

One artifact feeds several later tasks



Mixed CPU/GPU delayed reuse

Different artifacts compete for space



Recent hot set

Frequently reused artifacts fit in cache



Low reuse + revisions

Unique identities or changed content

Every trace runs through all four policies

Workload-aware | LRU | GPU-only LRU | Cost-only



Verify actual GPU pods
+ artifact correctness

20 matched trials · 80 policy arms · 1,280 MCP calls · 157,296-byte result budget

Controlled synthetic documents · Real MCP, Argo and GPU task execution

KUBERNETES EXPERIMENT FINDINGS

What Did We Actually Save?

Percent fewer GPU task-pod launches • Standard LRU = 0% • Higher is better

MEASURABLE SAVINGS

Mixed CPU/GPU delayed reuse

WORKLOAD-AWARE 27.3%

COST-ONLY · 9.1% fewer

GPU-ONLY LRU · 0%

High-fanout delayed reuse

AWARE 12.5%

COST-ONLY · 5.0% more

GPU-ONLY LRU · 0%

CONTROL WORKLOADS

Recent hot set

0% All four policies tied

Low reuse + revisions

0% All four policies tied

Five matched trials per family · Actual GPU task-pod launches, not measured cost or latency.

PRACTICAL DECISION GUIDE

When Is Workload-Aware Caching Worth It?

Does LRU evict expensive artifacts before their next valid reuse?

YES

WORTH EVALUATING

- **Expensive artifacts are reused**
- **Reuse happens much later**
- **Useful results compete for cache space**

Matched experiments: high-fanout + mixed delayed reuse

NO / RARELY

SIMPLER CACHE — OR NO CACHE

- **The working set already fits**
- **Requests rarely repeat**
- **Inputs change before valid reuse**

Matched controls: recent hot set + low reuse/revisions

Compare avoided recomputation with lookup, storage and coordination overhead.

OBSERVABILITY AND VERIFICATION

How Do You Know You're Actually Saving GPU Work?

Trace the same request across three independent layers

1 MCP Gateway

HIT / OWNER / WAITER / BYPASS

2 Argo Workflows

Workflow submitted—or not

3 Kubernetes GPU task

Pod UID + GPU request
Completion + validated output

New GPU work for this request?

GATEWAY	ARGO	KUBERNETES
GPU HIT	No new dispatch	No new GPU pod
MISS → OWNER	Submit workflow	Task + validated result
MISS → WAITER	No new dispatch	No additional GPU pod
CPU HIT	CPU reuse	Not GPU avoidance

Pod creation ≠ completion. Verify the result and successful execution.

Measure avoided GPU execution — not cache hit rate alone.

THREE LESSONS TO TAKE HOME

01

Understand reuse

Recently used does not always mean most valuable. Map downstream consumers and delayed reuse.

02

Reuse safely before dispatch

Validate equivalence before asking Argo and Kubernetes to compute again.

03

Measure actual GPU avoidance

Correlate MCP decisions, Argo workflows and Kubernetes GPU pods.



Before adding more GPUs, ask:

Are we recomputing work we have already done?

Agentics Day: MCP + Agents
NORTH AMERICA

Thank You

Stop Wasting GPUs

Workload-Aware Caching for Agentic AI Workflows on Kubernetes

What would this look like in your agent workflow?

SCAN FOR SLIDES

Chetan Sharma

Agentics Day: MCP + Agents · North America 2026

APPENDIX

Show Me the Code: Inside an MCP Cache Decision

```
async def embed_document(call):
    authorize(call)
    if not is_cacheable(call):
        return await execute_uncached(call)
    key = verified_identity(call)
    state = await cache.acquire(key)  # atomic
    if state.kind == "HIT":
        return authorized_result(call, state.value)
    if state.kind == "WAIT":
        return await wait_and_recheck(call, key)
    async with lease_scope(state):  # OWNER
        result = await argo.run_gpu_embedding(call)
        validate(result)
        await cache.commit_if_owner(state, result)
        return authorized_result(call, result)
```



Safe identity

Tool, source revision, model and scope.



Atomic acquisition

One decision:
HIT, WAIT or OWNER.



Verified commit

Validate output;
commit only as owner.

Illustrative API contracts, not literal source. Includes ownership checks; production still needs failure recovery and shared coordination.

APPENDIX

Cache Identity and Eligibility — Technical Details

Tool + version

Normalized operation identity

Arguments

Canonical request parameters

Source fingerprint

Content SHA + revision

Model + inference

Model fingerprint and settings

Implementation image

Code/runtime compatibility

Trusted scope

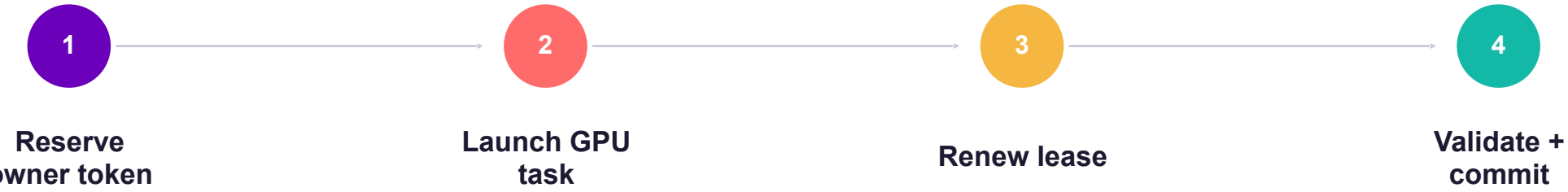
Authorization boundary

Cacheable examples: embed_document, immutable passage catalog

Bypass examples: live status, side-effecting record updates

APPENDIX

Single-Flight Execution and Failure Handling



WAITER PATH

Equivalent requests observe the in-flight owner, wait, and then read the committed artifact.

FAILURE PATH

Lease expiration allows takeover; ownership tokens prevent a superseded owner from committing stale output.

Leases protect ownership and commits—but do not guarantee exactly-once GPU execution under every failure.

APPENDIX

Retention Algorithm — Exact Study Configuration

$$S = D + 2C + 0.01F$$

Insertion-time score · Evict lowest first

D — **Scheduled immediate consumers**

Fixed plan count, not remaining tasks

C — **Measured compute seconds**

Excludes model load, scheduling and transport

F — **Cumulative tool-type invocations**

Snapshot at insertion; repeat polls deduplicated

Study configuration

157,296 serialized payload bytes · Fixed weights across trials · Same byte budget for all four policies

Study settings—not universal weights. Tie-break: recency, then cache key.

APPENDIX

Original Research — Absolute Latency Reference

Benchmark	Uncached	LRU	Workload-aware	vs LRU	vs uncached
MP-DocVQA	29.99s	22.35s	13.00s	**41.8%**	**56.7%**
SlideVQA	41.40s	29.93s	14.59s	**51.3%**	**64.8%**
VideoMME	40.05s	34.18s	27.38s	**19.9%**	**31.6%**

Headline claim: up to 51.3% lower average query latency than LRU (SlideVQA).

Full policy comparison and answer-quality metrics are in the paper.

Source: Workload-Aware Caching for Multi-Agent Systems · Table I · arXiv:2607.20495

APPENDIX

Methodology, Limitations and References

Environment

RTX 3070 · K3s · Argo Workflows
Python MCP SDK 1.30.0 · real GPU tasks

Study design

Fixed byte capacity · frozen traces and weights · paired policies · server-owned DAG metadata

Correctness

Parent hashes, revision identity, scope isolation, CPU consumers and GPU pod receipts reconciled

Evidence

Final archive SHA-256:
ddaee0afa7143e058d5c30e0003577b57
a6a7f4ce9e0fcf769b0e8d00d0666c3

Boundaries

Controlled synthetic traces; no measured production HA, autoscaling, GPU-hours, billing or pipeline latency.

References

Paper: arXiv:2607.20495
Implementation: frozen R1 experiment archive

Disclosure: R1+A1 amended the fitting-hot-set validator only; policy, capacity, traces and historical results were unchanged.