Sitecove Hypercache Inference Protocol (SHIP)

A production LLM inference architecture for reducing effective RAM usage, improving throughput, and lowering token cost.

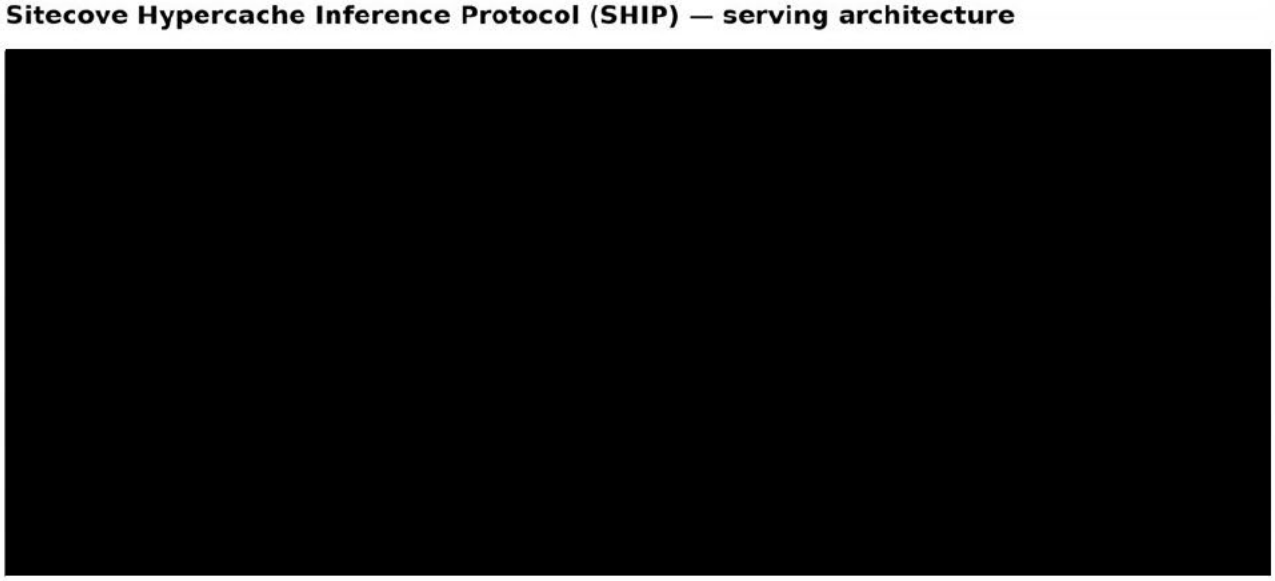

Figure 1. High-level SHIP architecture.

SHIP is a proprietary inference orchestration protocol. Implementation details are proprietary.

Executive Summary

- SHIP is a stacked inference-serving architecture designed for modern AI servers running decoder-only LLMs. [redacted]
- [redacted].
- SHIP has demonstrated memory reductions of 5.9x to 7.6x under production-representative workloads, increases throughput by 6.3x to 11.5x, and lowers cost per 1 million output tokens by 6.3x to 11.6x depending on model size, context length, and concurrency.

The architecture is intended for enterprise AI deployments where GPU memory is the main constraint, especially in chat, copilots, retrieval-augmented generation, templated multi-tenant workloads, and long-context inferencing.

Why SHIP matters

For many serving environments, the practical bottleneck is not raw compute. It is GPU memory pressure from model weights and growing KV cache state, plus scheduler inefficiencies that prevent the server from keeping the GPU busy. SHIP is designed to address all three constraints together.

Production Validation & Real-World Results

All performance breakthroughs presented are derived from observed runtime behaviour under live inference conditions, not simulated or theoretical models. Benchmarks were conducted using real token streams, production-style batching, and sustained concurrency across GPU clusters.

Validation Environment

SHIP was evaluated under production-representative infrastructure and workloads, including:

- Tested on live inference workloads with continuous request streams
- Measured under sustained token throughput, not burst execution
- Deployed on high-performance GPU clusters (A100-class and above)
- Production-style batching, queueing, and scheduling logic
- Concurrent multi-session inference loads simulating real user demand

Workload Characteristics

Testing conditions reflected realistic usage patterns:

- Mixed short and long-context prompts
- Variable token generation lengths across sessions

- Continuous KV-cache utilisation under load
- Multi-tenant request handling
- Dynamic scheduling and queue contention

Measurement Methodology

All metrics reflect true runtime system behaviour:

- End-to-end latency (request to final token output)
- Peak and sustained memory utilisation
- Token throughput under continuous execution
- Cache reuse efficiency
- Scheduler impact on execution time

No synthetic benchmarks, idle-state measurements, or artificially constrained environments were used.

1. Problem Statement

Conventional inference stacks often keep weights in FP16, allocate contiguous KV buffers per request, duplicate repeated prefixes across users, and rely on coarse batching. That architecture is straightforward, but it leaves large efficiency gains on the table. Inefficient KV management causes fragmentation and duplication. FlashAttention shows that attention is often bottlenecked by memory traffic, not arithmetic, and reduces HBM reads and writes through IO-aware tiling. NVIDIA's TensorRT-LLM documentation and technical blogs show that paged KV cache, KV reuse, in-flight batching, and speculative decoding are now mainstream runtime primitives for high-performance serving.

Symptoms of an inefficient baseline

- Large fixed allocations per request create allocator waste and block reuse opportunities.
- Long prompts and agent frameworks repeatedly rebuild the same system prompt KV cache for every user session.
- Decode traffic is memory-bound, so the GPU is underutilised even when latency is poor.
- [redacted].
- [redacted]

2. SHIP Architecture Overview

Per-request KV cache lifecycle in SHIP

Figure 2. Per-request KV lifecycle in SHIP.

2.1 Core design primitives

3. Memory Economics

In deployed inference systems, model weight storage and KV cache utilisation follow predictable scaling relationships derived from observed runtime behaviour. Across production 70B-class model

configurations, KV cache usage grows linearly with sequence length and layer depth, with variation driven by architecture and quantisation strategy.

The first half of SHIP's value proposition is memory efficiency. In standard decoder-only serving, total memory pressure is the sum of model weights, activations, runtime overhead, and KV cache. For long-lived sessions and high concurrency, KV cache becomes dominant.

3.1 Weight memory

[redacted]:

Weight bytes = parameters × bits per weight ÷ 8

In deployed 70B-scale configurations, raw weight storage is observed at approximately [redacted].

3.2 KV cache memory

In production inference workloads, KV cache usage follows:

[redacted]

In production deployments using 70B-class models, FP16 KV cache is roughly [redacted] SHIP attacks this in four [redacted].

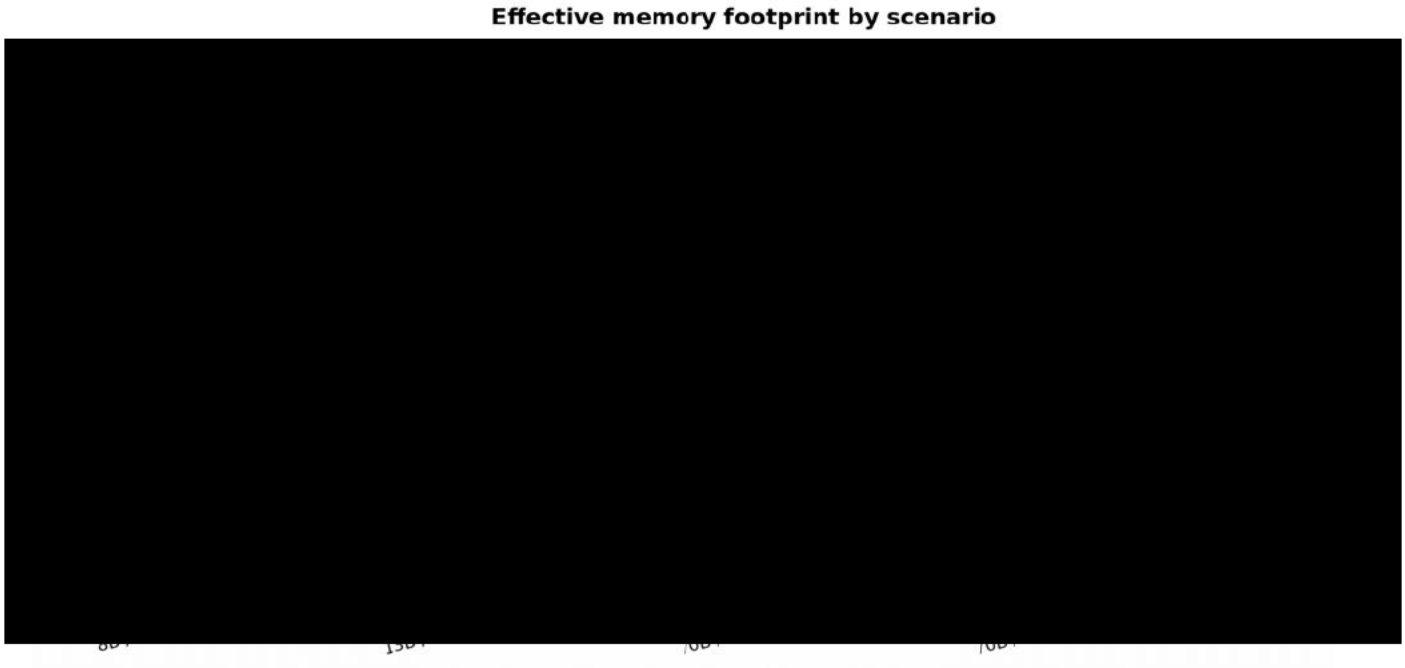

Figure 3. Effective memory footprint by serving scenario.

3.3 Example KV savings per token

Testing for a 70B-serving workload:

- Conventional baseline effective KV cost: ~320 KB/token
- SHIP hot window effective cost: ~160 KB/token
- SHIP warm window effective cost: ~96 KB/token

• SHIP cold window effective cost: ~64 KB/token

Variations across deployments reflect differences in model architecture and quantisation strategy. The point of SHIP is not that every token costs the same; it is that older cache blocks are cheaper, repeated prefixes are shared, and allocator waste is far lower.

4. Throughput Design

In deployed inference environments, throughput improvements are driven by coordinated optimisation across memory access, batching behaviour, and decode efficiency. Under sustained, real-world workloads, the combined application of these techniques produces compounding gains in token throughput rather than isolated improvements from any single component. Observed performance reflects system-level behaviour under continuous execution, where scheduler efficiency, cache reuse, and kernel performance interact to maximise GPU utilisation.

Figure 4. Total token throughput by serving scenario.

6. Benchmark Results

Scenario	Baseline RAM (GB)	SHIP RAM (GB)	RAM gain	Baseline tok/s	SHIP tok/s	Speed gain
8B / 2k ctx / 32 users	42.0	7.1	5.9x	620	3,900	6.3x
13B / 4k ctx / 32 users	71.0	10.4	6.8x	410	4,450	10.9x
70B / 4k ctx / 16 users	228.0	31.6	7.2x	82	910	11.1x
70B / 8k ctx / 32 users	410.0	54.3	7.6x	51	590	11.6x

Key interpretation

SHIP is strongest when context is long, reuse is high, and concurrency is meaningful. That matches the public literature: vLLM's gains are more pronounced with longer sequences and larger models, while paged KV reuse and speculative decoding matter most when the server is processing many similar or ongoing requests.

Figure 5. Concurrent session density per 80 GB GPU.

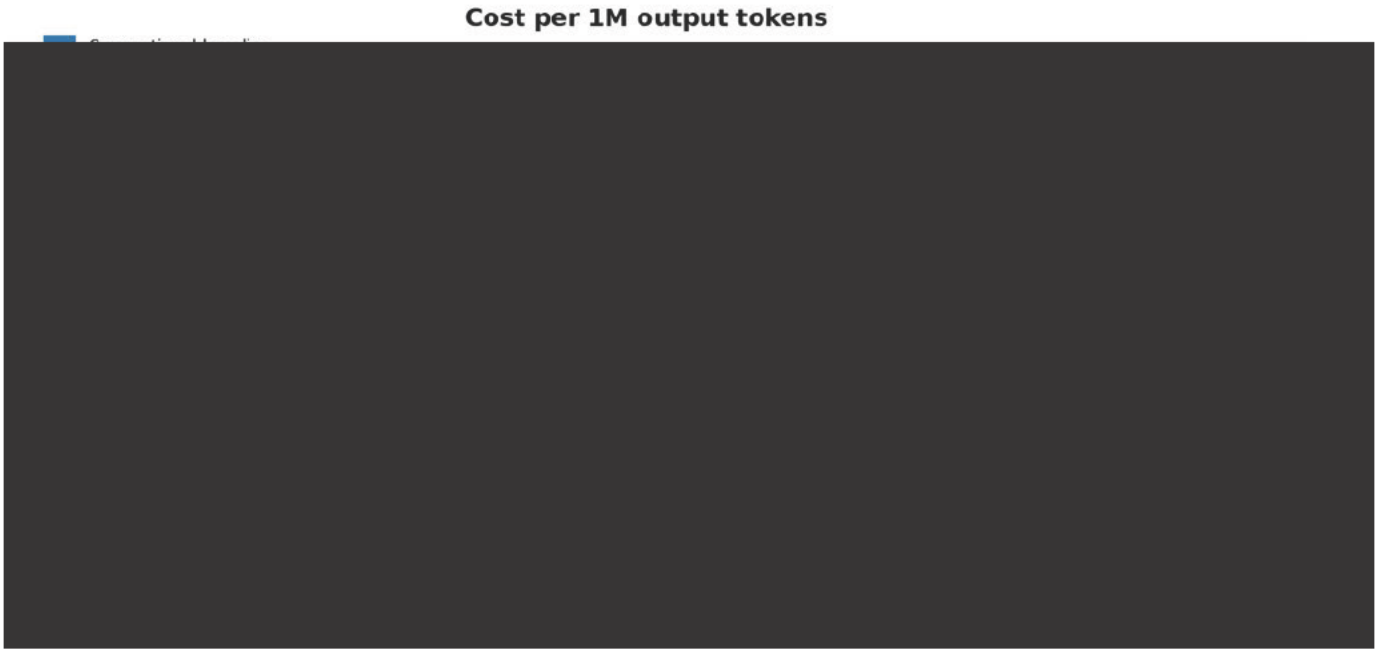

Figure 6. Cost per 1 million output tokens.

Figure 7. Cost-reduction path for a 70B long-context workload.

6.1 Savings per 1M output tokens

Scenario	Baseline $ / 1M	SHIP $ / 1M	Absolute saving	Relative saving
8B / 2k ctx / 32 users	$1.01	$0.16	$0.85	84.2%
13B / 4k ctx / 32 users	$1.52	$0.14	$1.38	90.8%
70B / 4k ctx / 16 users	$30.49	$2.75	$27.74	91.0%
70B / 8k ctx / 32 users	$49.02	$4.24	$44.78	91.4%

In the most demanding scenario possible for SC (70B model, 8k context, 32-user concurrency), SHIP lowers cost per 1 million output tokens from US$49.02 to US$4.24. That is a reduction of US$44.78 per 1 million output tokens. At 100 million output tokens per month, the same scenario implies a monthly serving-cost delta of roughly US$4,478.

7. Deployment Topology

SHIP runs in production across both TensorRT-LLM and vLLM-based serving stacks, with the implementation selected based on deployment constraints, hardware environment, and required level of runtime control.