# The Anatomy of a Prompt

## Why Prompt Anatomy Becomes Storage Anatomy in Long-Context Inference and RAG

Jeff Harthorn
AI Applied Research Lead

SOLIDIGM

# Abstract

Prompt structure deterministically materializes state objects (e.g., KV cache blocks, retrieval payloads, and tool artifacts) that drive storage architecture, qualification, and QoS requirements for LLM inference and RAG. This white paper provides a mapping artifact (see Table 1) that connects prompt layers to state objects, I/O patterns, dominant SLOs, and tier placement decisions, reframing "prompt engineering" as an infrastructure discipline. It is written for infrastructure and platform teams who must design, architect, and qualify AI serving stacks — and specifically to help them understand where storage enters the serving path and how to qualify it alongside compute, memory, and network. Two design and qualification implications follow:

1. Long context and agentic workflows turn "GB per session" into "TB per fleet" through KV scaling under concurrency, and
2. RAG behaves like a low-queue-depth random-read workload whose p99/p99.9 latency, not headline IOPS at high queue depth, gates end-user time-to-first-token.

# Executive Summary

A production prompt is a compiled context bundle: policies, templates, user intent, session history, retrieved evidence, tool schemas, and tool outputs. For a given model and serving stack, prompt structure determines which state objects are instantiated (KV cache blocks, retrieval payloads, tool artifacts) and how they scale. Those state objects impose measurable I/O shapes and QoS requirements. Context windows are expanding (some models expose 1M-token windows), increasing prefill work and state footprint.[1] Long context and agentic loops increase state; concurrency multiplies it.

KV cache becomes the dominant in-flight working set and a first-order limiter for TTFT, ITL stability, and cost.[2,3,4] RAG behaves like a low-queue-depth random-read workload under concurrency. Procurement success is defined by p99/p99.9 latency at QD1–4 under realistic concurrency and mixed load, not peak IOPS or peak GB/s at high queue depth.[5,6,7] If your prompts include large stable prefixes, grounding rules, tool schemas, and retrieval payloads, you are implicitly choosing a tiering design and a tail-latency requirement under concurrency. Table 1 (§2) makes that mapping explicit. §8 converts it into a procurement test plan

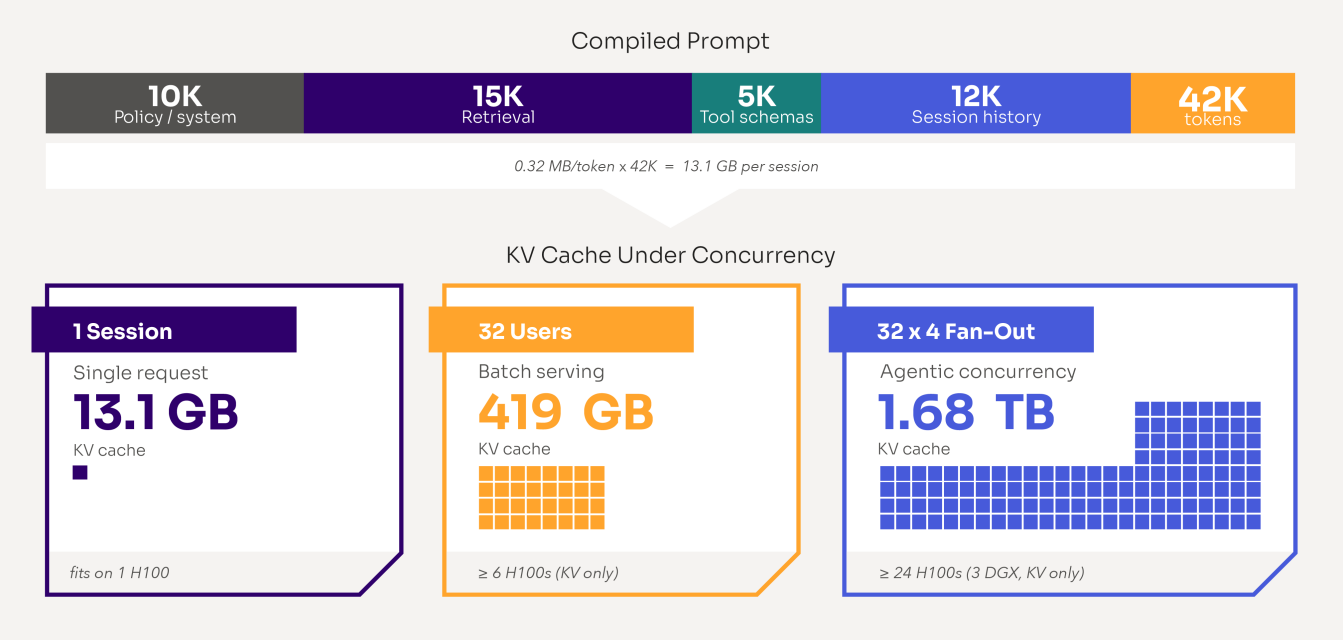


*Figure 1. One prompt, terabytes of state. Note: KV anchored on NVIDIA 40 GB @ 128K tokens (Llama 3 70B, FP16).[2] 42K prompt = illustrative governed-assistant build-up (§0). Expanded in Figures 5-6.*

# Table of Contents

1 **The reframing:** Inference is stateful and prompts materialize state ..... 4

2 **Table 1:** From prompt anatomy to storage anatomy ..... 5

3 **Stable-early, volatile-late:** A practical prompt layout rule ..... 6

4 **The prompt-to-I/O path:** TTFT vs ITL and where storage enters ..... 7

5 **RAG is a storage workload:** qualifying for p99 at QD1–4 ..... 9

6 **KV cache scale and the concurrency cliff** ..... 12

7 **A practical storage hierarchy for inference state** ..... 17

8 **Procurement-grade qualification checklist** ..... 20

9 **Operational and security implications** ..... 22

10 **Conclusion** ..... 23

Appendix A. KV cache approximate sizing cheat sheet ..... 24

Appendix B. RAG I/O envelope by architecture (illustrative) ..... 25

Appendix C. Benchmarking checklist for RAG reality ..... 26

Appendix D. Glossary ..... 26

References ..... 27

# 1. The reframing: Inference is stateful and prompts materialize state

Enterprises increasingly deploy LLMs as assistants that must remain grounded, governed, tool-connected, and responsive. In that setting, prompts are not freeform strings typed by users. They are compiled context bundles, assembled at request time by orchestrators that draw from multiple governed sources: stable policy prefixes, session history, retrieved evidence, tool schemas, and tool outputs. The compilation is repeatable; its structure is predictable, and its downstream effects on state and I/O are measurable. Storage belongs in the same design conversation as compute, memory, network, and orchestration: prompt structure shifts pressure across the whole serving path, and planning for one dimension in isolation invites silent failure modes.

This paper treats "prompt engineering" as an infrastructure discipline. Prompt structure is a primary input to storage architecture and QoS procurement because it determines the classes of state your serving stack must create, move, and retain.

## 1.1 Scope and precision

This is not a vector database feature comparison or benchmark report. Several behaviors (e.g., KV paging/offload, prefix reuse, restore timing) are implementation-dependent. Where behavior depends on implementation, we label assumptions explicitly and convert claims into operator guidance. Where a statement cannot be supported publicly, it is phrased as "operators should measure …" rather than as a universal claim.

## 1.2 Canonical workload persona = "Joe"

Consider a regulated enterprise code-assistant deployment. Joe, a staff engineer, uses an IDE-integrated assistant connected to internal repositories, ticketing, CI, and documentation. Governance requires that tool interactions and outputs are retained, access-controlled, and auditable. Each request becomes a workflow: 1) Retrieve evidence, 2) Load tool schemas, 3) Run tools, 4) Write artifacts, and 5) Iterate. The unit of work is not one prompt and one answer, it is a multi-step loop whose intermediate artifacts become both model-visible state and compliance evidence.

We will return to Joe's workflow throughout this paper:

- §4 when his retrieval calls create the low-QD random-read storm
- §6 when his compiled prompts drive the 42K-token build-up
- §9 when his audit trail expands the write plane

## 2. Table 1: From prompt anatomy to storage anatomy

Table 1 is the paper's primary mapping artifact: each prompt layer materializes state objects with recognizable I/O signatures, and those signatures map to tier placement rules and qualification criteria. Read it left-to-right as a five-step translation from prompt component through state object, I/O shape, and SLO risk to a tiering and qualification implication.

| Prompt layer / component | Materialized state objects | Typical serving-time I/O shape (illustrative) | Dominant SLO risk | Tiering & qualification implication |
|---|---|---|---|---|
| **Context (system, policy, history)** | Stable prefixes; session history; optional prefix-KV for prefix reuse | Mostly memory-resident; restore reads if persisted across hosts | TTFT and cost (prefill); cache hit rate stability | Keep templates hot in DRAM; if persisted, qualify restore latency.[8,9] |
| **Few-shot / exemplars (optional)** | Prompt library objects; optional prefix-KV | Read-mostly, infrequent updates | TTFT/cost and reuse efficiency | Treat as stable prefix; cache near compute; avoid co-location with churn-heavy write planes. |
| **Grounding (constraints, citations, guardrails)** | Retrieved spans + citation/constraint metadata | Random reads (metadata + payload) with fanout; block sizes depend on packing | p99 retrieval latency (translates to p99 TTFT); correctness/ compliance | Optimize p99/p99.9 at QD1–4 under concurrency; keep citation metadata hot; isolate from write planes.[5,6,7] |
| **RAG (retrieval stack)** | Vector index + chunk store | Low per-thread QD random reads under high concurrency; straggler amplification with sharding | Tail latency and stragglers dominate TTFT | Qualify many concurrent jobs at QD1–4; measure p99/p99.9 under mixed load; replicate/hedge hot shards where applicable.[5,10] |
| **Tool schemas / definitions** | Tool registry; selected schemas; policy enforcement metadata | Read-mostly bursts during tool discovery; risk is token growth | TTFT and token budget explosion | Load on demand and cache near compute; avoid inlining entire catalogs.[11] |
| **Tool outputs (logs, diffs, test results)** | Artifact store; audit traces; derived caches | Write bursts; debug readbacks; mixed R/W with background maintenance | Deterministic QoS under mixed load (read tails, jitter) | Higher-endurance tier; isolate from retrieval read planes.[7] |

*Table 1. Primary mapping artifact. Note: I/O shapes and block sizes are illustrative and implementation-dependent; qualify with workload-representative tests.*

## 3. Stable-early, volatile-late: A practical prompt layout rule

Table 1 reveals a natural ordering principle that has direct operational consequences. Stable components (e.g., system policies, templates, few-shot exemplars) change rarely across requests, while volatile components (e.g., retrieval payloads, tool outputs, recent conversation turns) change every time.

The operational heuristic is stable-early, volatile-late. Place stable policy and template segments at the beginning of the prompt to enable prefix reuse and append volatile segments (e.g., retrieval payloads, tool outputs) at the end to reduce cache churn and isolate variability.

This layout directly affects three things:

1. **Cacheability:** Prefix caching refers to reusing previously computed KV state for prompt segments that are identical across requests. We call this prefix reuse throughout. Prefix-caching systems, including those documented by OpenAI,[8] match from the beginning of the prompt. A stable early layout maximizes the probability that prior KV computations can be reused, avoiding redundant prefill work.

2. **Prefill cost:** When a prefix match succeeds, the serving stack skips prefill for the matched tokens and only processes the volatile suffix. The longer the stable prefix relative to the full prompt, the greater the savings.

3. **Restore probability:** If the prefix KV is evicted and must be restored from a lower tier (e.g., DRAM, NVMe, or network), only the suffix requires fresh computation. The restore cost is bounded by the prefix length, and the compute cost is bounded by the suffix length.

Preble reports that across five public workloads—including tool use, programming, video QA, and long-document QA—roughly 85% to 97% of prompt tokens are shared with other requests within each workload, which is exactly the condition under which early, stable prompt regions are worth preserving for reuse.[13] SGLang's cache-aware routing results show the operational consequence. When prefix locality is preserved, prefix-cache hit rate rises from 20% to 75%, with a corresponding throughput gain.[14] Taken together, these results support the claim that placing durable instructions, schemas, and policy text early in the prompt is a measurable systems optimization that reduces redundant prefill work and improves cache effectiveness.

For Joe (§0), this means his IDE assistant's orchestrator should assemble the prompt as: compliance policy, tool allowlist, few-shot exemplars, retrieved code context, recent conversation history, current user query. Stable segments are first, volatile segments last. The first three are fixed across a session or across sessions while the last three change per request. This ordering maximizes the reusable prefix.

How much that reuse saves depends on where latency accumulates in the inference path, which is the subject of the next section.

# 4. The prompt-to-I/O path: TTFT vs ITL and where storage enters

Interactive inference has two user-visible latency regimes that behave differently under load. Time-to-first-token (TTFT) is the delay before the model begins responding. It is dominated by prompt assembly (i.e., retrieval + tool I/O + compilation) and by prefill. Inter-token latency (ITL) is the per-token "typing" cadence during decode. It's smooth when KV remains resident and becomes visibly jittery when required state must be restored on-demand.

This section makes the storage contract explicit. Storage most often enters the critical path through 1) retrieval and tool data-plane latency during prompt assembly and, 2) restore latency for reusable prefixes or evicted KV state. Procurement and architecture should therefore treat TTFT and ITL separately. Optimize TTFT by reducing p99/p99.9 tails in the assembly path and protect ITL by ensuring any restores are pre-staged before decode.

### 4.1 Prefill versus decode: Operational split

We separate two phases because they have different storage sensitivities. Prefill ingests the full input sequence and writes KV vectors for prompt tokens into the KV cache. Decode generates output tokens iteratively by repeatedly reading prior KV and appending new KV. KV caching is a standard optimization for autoregressive decoding.[3]

As long as the KV working set remains resident in GPU high-bandwidth memory (HBM) and/or staged in DRAM for paging designs, storage should not be on the steady-state decode critical path. When KV state must be restored from a lower storage tier, on-demand faults during decode can introduce ITL jitter. Operators should pre-stage the required KV blocks back into memory before decode begins, treating restore-before-decode as an explicit design objective.

This is the governing rule for ITL stability. If restore happens during decode, storage tail latency becomes token-rate jitter.

### 4.2 TTFT decomposition: Procurement view

A useful decomposition for procurement and profiling separates the full critical path into five observable phases:

$$TTFT = T_{\text{network}} + T_{\text{queue}} + T_{\text{assembly}} + T_{\text{prefill}} + T_{\text{decode}_1}$$

where:

- $T_{\text{network}}$ is round-trip latency between the client and the API gateway, approximately constant for a given deployment; typically 5–80 ms depending on geography.
- $T_{\text{queue}}$ is time spent waiting in the serving scheduler's request queue. It is load-dependent; near-zero under low utilization, dominant under saturation.
- $T_{\text{assembly}}$ is the time to compile the prompt: Retrieval calls, tool I/O, schema loading, and orchestrator logic. Under fanout, end-to-end assembly latency is increasingly dictated by the slowest component (i.e., the "tail at scale" effect).[5]

- $T_{\text{prefill}}$ is the compute-bound phase that processes the full input sequence and materializes KV cache. $T_{\text{prefill}}$ grows with prompt length. For exact attention, attention work remains quadratic in sequence length, while optimized kernels reduce constants and memory traffic.
- $T_{\text{decode}_1}$ is the first decode step. It's memory-bandwidth-bound, approximately O(1) with respect to prompt length.

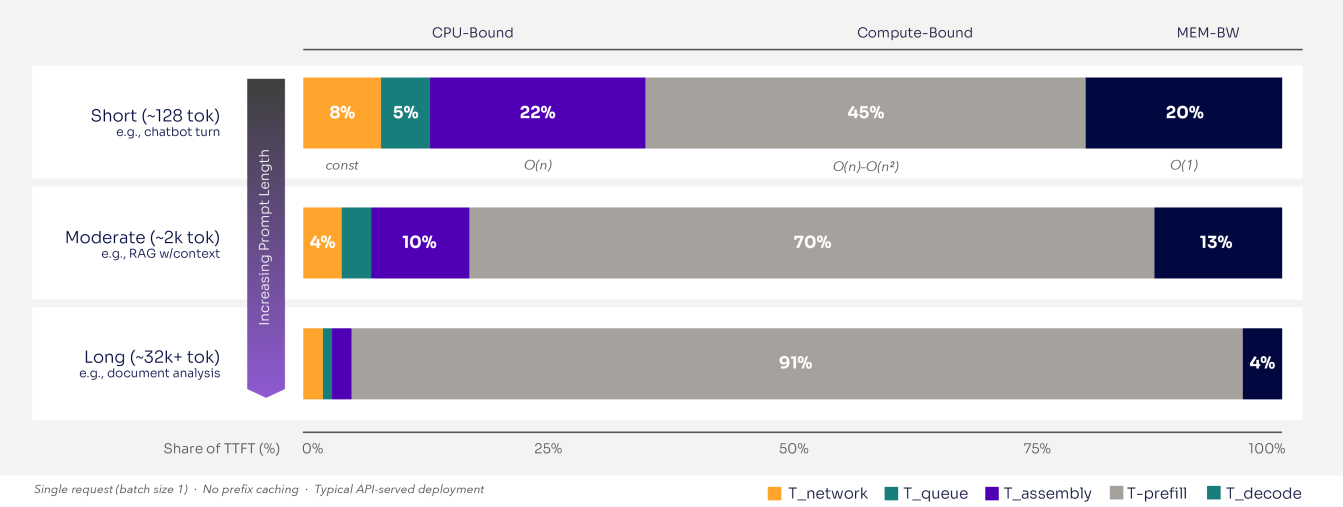


*Figure 2. Breakdown of Time to First Token across prompt-length regimes.[15,16,17,18,19,20] Note: Proportions are representative estimates for batch-size-1, no-prefix-caching scenarios. Actual ratios vary by model, hardware, and load.*

A recent RAG inference study reports that retrieval alone accounts for roughly 45% to 47% of TTFT in its evaluated setups and gives a representative example in which TTFT rises from 495 ms to 965 ms when retrieval is added.21 This supports the inclusion of an "assembly" phase rather than treating TTFT as an LLM-only quantity. NVIDIA Inference Microservices (NIM) benchmarking documentation is consistent with that framing, defining TTFT to include request queuing, prefill, and network latency in addition to model execution.16

The five-term decomposition matters for procurement because different terms are gated by different subsystems. $T_{\text{network}}$ and $T_{\text{queue}}$ are infrastructure and scheduling problems. $T_{\text{assembly}}$ is where storage enters through retrieval and tool I/O. $T_{\text{prefill}}$ is compute-bound but influenced by prefix caching which may involve storage restores. $T_{\text{decode}_1}$ is memory-bandwidth-bound and largely storage-independent when KV is resident.

### 4.3 Where storage enters the critical path

Storage most commonly gates TTFT through two mechanisms:

1. **Assembly-path latency:** Every retrieval call, tool-schema load, and artifact readback during prompt compilation is a storage I/O. Under scatter-gather retrieval, the assembly phase is bounded by the slowest shard's response.
2. **Prefix restore latency:** When a stable-early prefix has KV cache that was evicted to a lower tier like DRAM, NVMe, or network, restoring it adds to $T_{\text{prefill}}$. If restore is faster than recomputation, the net effect is a TTFT reduction. If restore is slower due to storage tail latency or contention, it becomes a TTFT penalty.

For Joe's IDE assistant that we introduced in §0, the assembly phase includes: (a) a vector-similarity search against his team's code repository, (b) fetching the top-k code chunks from the chunk store, (c) loading the selected tool schemas (e.g., linter, test runner, CI status), and (d) any readback of prior tool outputs. Each of these is a storage I/O, and the slowest one sets the floor for $T_{\text{assembly}}$.

## 5. RAG is a storage workload: qualifying for p99 at QD1–4

RAG is not “just search.” Retrieved passages are injected into the prompt token stream, so retrieval often gates when prefill can begin. In sharded systems, scatter-gather retrieval amplifies tail latency and completion time tends toward the slowest shard.[5,10]

### 5.1 The random-read storm model: What to measure

Under concurrency, we should treat retrieval as a QoS workload. The defining characteristic is this: Per-thread queue depth is low at QD1–4, but many independent sessions generate high aggregate outstanding I/O across multiple submission queues.

To make this concrete, consider Joe's deployment at moderate scale:

- 200 concurrent user sessions, each issuing a retrieval request.
- Each retrieval fans out across 3 shards. A typical HNSW or IVF-PQ partition count for a mid-size corpus.
- Each shard query issues QD=2 of outstanding I/O, with one index probe + one chunk fetch, or two sequential chunk fetches.

The device sees:

$$200 \text{ sessions} \times 3 \text{ shards} \times \text{QD2} = 1{,}200 \text{ outstanding I/Os}$$

across ~600 submission queues. But each ***individual thread*** still sees QD=2. The per-thread latency distribution, not the aggregate throughput, is what determines whether Joe sees a fast or slow TTFT.

This is why peak IOPS at QD=128 for a single thread is an incomplete procurement metric. It might show the shape of the performance, but it measures a regime that doesn't exist in this workload. What matters is p99 latency at QD=1–4 when hundreds of independent threads are concurrently submitting. Procurement-grade qualification should therefore use many concurrent jobs at low per-job QD and report p99/p99.9 latency under steady-state and mixed-load conditions.[6]

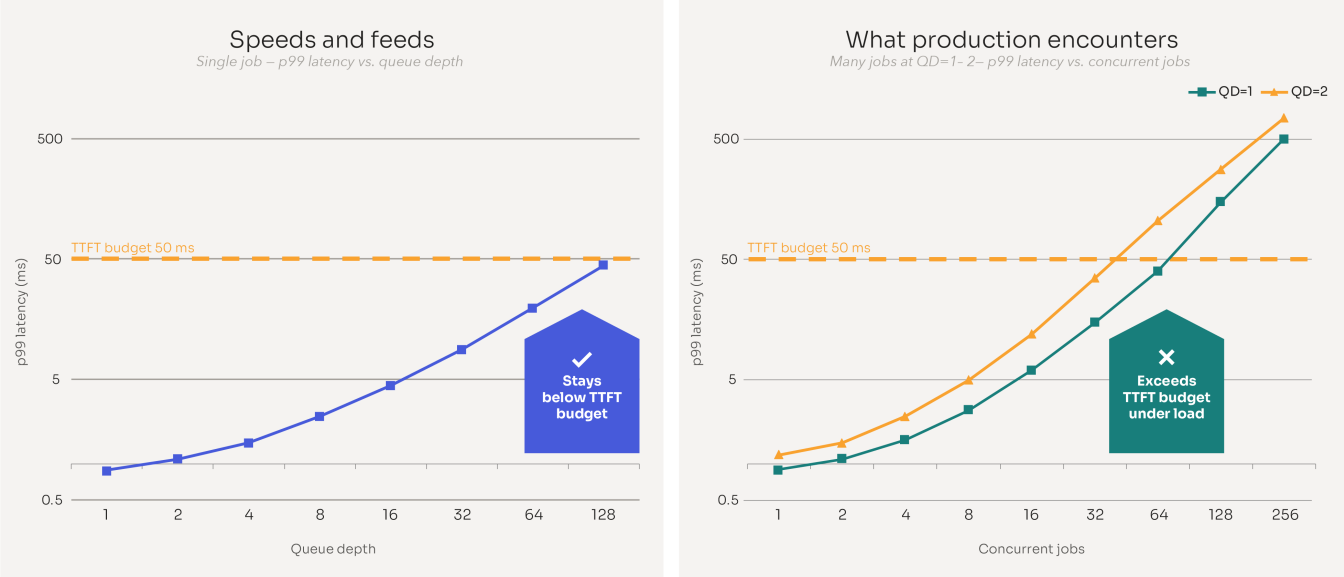


*Figure 3. The concurrency cliff: Tail latency at low queue depth. Note: Illustrative curves; not measured data. Shapes derived from published NVMe/storage concurrency behavior. See Appendix C for procurement test methodology. TTFT budget line = assembly-phase latency allocation within total TTFT.*

The argument that many low-QD issuers can be worse for tail latency than one deep issuer is supported by recent scheduler characterization work on NVMe SSDs.

Ren et al. explicitly compare concurrency created by increasing queue depth within one process against concurrency created by many concurrent QD=1 processes and show materially worse tail behavior in the inter-process case. As concurrency rises, inter-process p99 latency is reported at roughly 4x intra-process p99, including a concrete Kyber example of 1,003.5 µs versus 4,227.1 µs at an aggregate concurrency level of 256.[22]

Intel's foundational NVMe benchmarking guidance reinforces the same evaluation discipline by emphasizing queue-depth operating region and high-percentile QoS, noting that latency rises with queue depth and that 99.99th-percentile latency is more meaningful than average latency for outlier-sensitive applications.[6]

That combination makes the procurement thesis much stronger. Realistic inference storage pressure must be judged by p99/p99.9 behavior under many independent misses, not by peak IOPS at synthetic deep queue depth.

### 5.2 Mixed-load interference

Mixed-load interference matters. SSD internal maintenance, including garbage collection, can introduce long-latency events that become visible to users as TTFT spikes when latency-critical reads share devices with churn-heavy writes.[7]

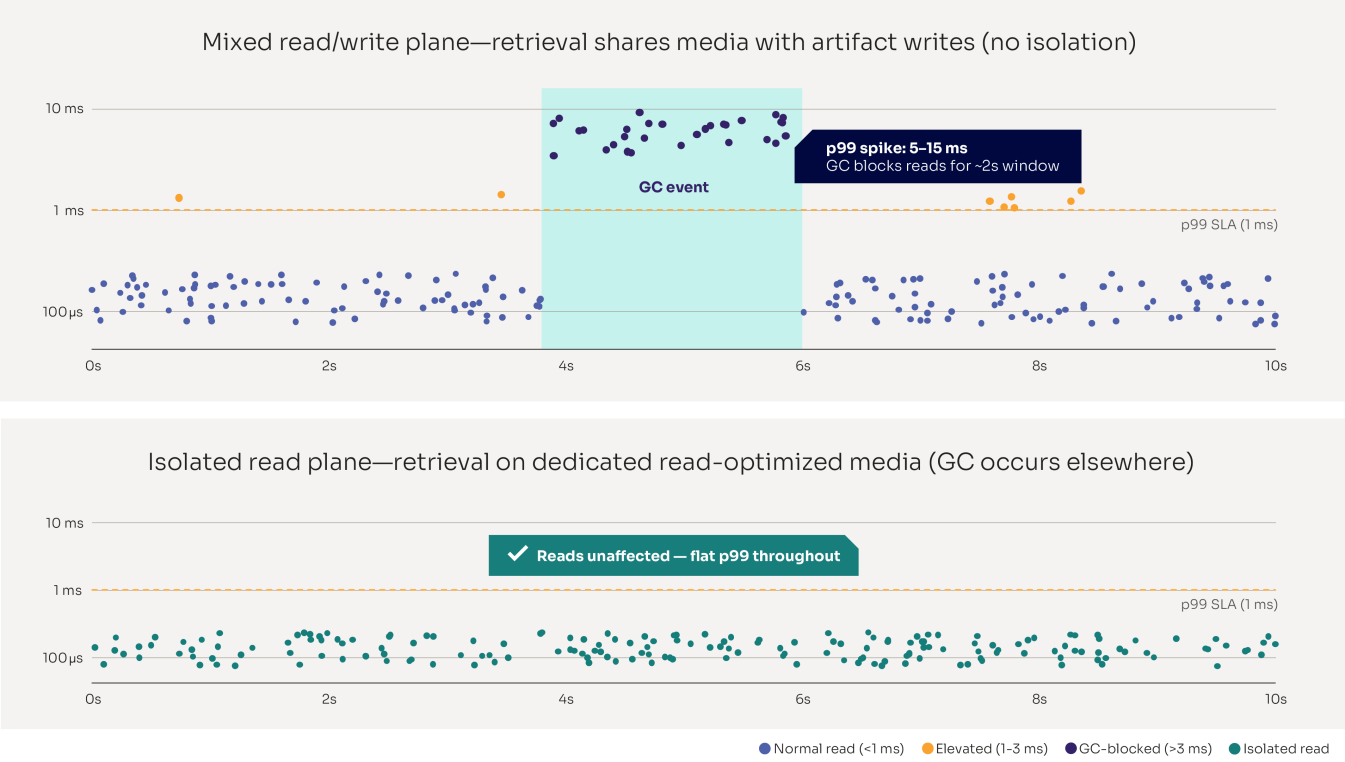


*Figure 4. GC interference - Why isolation determines tail latency. Note: Illustrative timeline; not measured data. GC spike magnitude and duration are representative of published NVMe/flash behavior under sustained mixed workloads. See §0 for placement policy details.*

### 5.3 Mitigation levers: Storage + orchestration

- Replicate hot shards and route requests to the lowest-latency replica when feasible.
- Use hedged reads when shard latency exceeds a percentile threshold, with aggressive cancellation to limit overhead. This is implementation-dependent.
- Co-locate index and chunk-store shards when it reduces fabric tails, or explicitly budget network p99/p99.9 if disaggregated. This is implementation-dependent.
- If disaggregating storage, account for fabric effects and protocol overhead, for example NVMe-oF.[23]
- Isolate artifact/log write planes from retrieval read planes so background maintenance cannot poison retrieval p99.

## 6. KV cache scale and the concurrency cliff

KV cache footprint grows approximately linearly with (a) tokens processed and (b) the number of active sequences.[2,4] This is why long-context inference can hit a memory wall even when model weights fit. Weights are mostly static, while KV grows with usage and concurrency.

### 6.1 From governed assistant to ~42K tokens: A build-up

Enterprise assistants are not minimal prompts. They are compiled context bundles that accumulate tokens from multiple governed components. The following illustrative build-up shows how a ~42K-token prompt is operationally plausible in production environments, using publicly documented observations.

1. **Stable system and policy context: ≈5K to 22K tokens**

   WEKA reports that system prompts in real deployments commonly range from ~5,000 to ~22,000 tokens before retrieval payloads or tool outputs are added.[9] In regulated or policy-heavy environments, this stable context can include compliance rules, domain instructions, formatting constraints, and safety guardrails. This example assumes ~10K tokens for stable prefix.

2. **Retrieval payload: ≈10K to 20K tokens**

   Retrieval-augmented generation injects external evidence into the prompt.[10] Depending on chunking strategy and context window size, it is common to include 10 to 20 chunks. If each chunk contributes on the order of 500 to 2,000 tokens, retrieval alone can add ~10K to 20K tokens. Google's long-context documentation equates 1M tokens to roughly 50,000 lines of code, illustrating how large context windows enable substantial evidence injection.1 We assume ~15K tokens.

### 3. Tool schemas and definitions: Variable, potentially large

Tool definitions are a first-order context-budget concern. Anthropic reports that a five-server, 58-tool stack consumes approximately 55K tokens before conversation begins, and that unoptimized tool catalogs have reached 134K tokens in internal experience.[11,24] Aggressive optimization such as on-demand loading or execution-environment processing can reduce this to roughly 2K tokens. OpenAI's function-calling guide describes the same pressure, recommending fewer than 20 functions at the start of a turn and deferred loading for the rest, and noting that all callable definitions count against the context limit and are billed as input tokens.[25] Even after optimization, selected tool schemas and policy metadata typically add several thousand tokens to a compiled prompt when multiple tools are active. For the build-up, we'll assume moderate optimization to ~5K tokens.

### 4. Conversation history: Multi-turn accumulation

Multi-turn enterprise assistants accumulate dialogue history over a session. With extended context windows now common,[1] thousands of additional tokens of user intent, clarifications, and prior outputs can persist in context. Estimate this to be ~12K tokens for this example.

This is neither baseline nor maximum, merely a realistic mid-range example derived from publicly observed deployment patterns.[2,9,12,26]

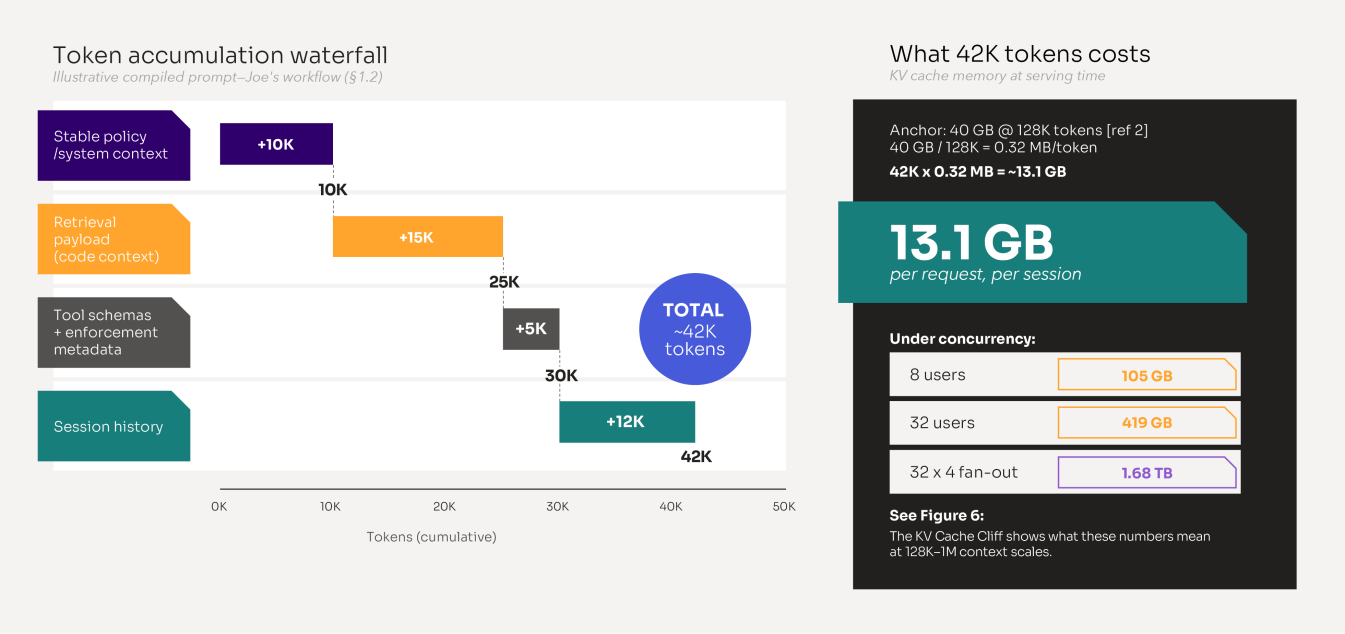


*Figure 5. From Governed Assistant to ~42K Tokens. Note: Illustrative build-up for a governed enterprise assistant (Joe's workflow, §0). Token counts are representative mid-range values from KV anchor: NVIDIA 40 GB @ 128K (Llama 3 70B, FP16).*[1,2,9,10,11,12]

### 6.2 What 42K tokens costs in memory

Anchoring to NVIDIA's published figure of ~40 GB of KV cache for a 128K-token context on Llama 3 70B:[2]

$$\frac{40\ \text{GB}}{128{,}000\ \text{tokens}} \approx 0.32\ \text{MB/token}$$

$$42{,}000\ \text{tokens} \times 0.32\ \text{MB/token} \approx 13.1\ \text{GB}$$

The implication is structural. Long-context enterprise prompts transform "gigabytes per session" into terabytes under concurrency. KV scale is therefore not an edge case; it is a predictable outcome of governed, retrieval-connected assistant design.[3,4,5]

### 6.3 A concise KV scaling table

Long context turns "GB per session" into "TB per fleet" quickly. This is why serving stacks increasingly treat KV as a managed resource—paged, reused, and sometimes externalized across tiers.[4]

| Context length | KV/seq (Llama 3 70B, 40 GB @ 128K)[2] | | | KV/seq @ 100 conc. | | | KV/seq @ 1,000 conc. | | |
|---|---|---|---|---|---|---|---|---|---|
| | FP16 | FP8 | FP4 | FP16 | FP8 | FP4 | FP16 | FP8 | FP4 |
| 42K tokens | ~13.1GB | ~6.6GB | ~3.3GB | ~1.3TB | ~0.6TB | ~0.3TB | ~12.8TB | ~6.4TB | ~3.2TB |
| 64K tokens | ~20.0GB | ~10.0GB | ~5.0GB | ~2.0TB | ~1.0TB | ~0.5TB | ~19.5TB | ~9.8TB | ~4.9TB |
| 128K tokens | ~40.0GB | ~ 20.0GB | ~10.0GB | ~3.9TB | ~2.0TB | ~1.0TB | ~39.1TB | ~19.5TB | ~9.8TB |

*Table 2. Context length. Note: assumes linear scaling from NVIDIA's published 40GB@128K anchor and is a back-of-envelope sizing aid; validate with stack telemetry. FP8/FP4 reduce memory footprint but have quality trade-offs.*

TensorRT-LLM documents explicit support for FP8 KV cache quantization.[27] As a first-order capacity estimate, moving KV storage from a 16-bit format to FP8 halves bytes per element and therefore doubles effective KV headroom at a fixed memory budget. NVIDIA's NVFP4 KV cache guidance extends that logic by positioning NVFP4 as a further halving relative to FP8, which makes an approximate 4x-versus-FP16/BF16 capacity framing reasonable for an illustrative table.[28]

This is operationally meaningful, but it does not change the underlying scaling law: KV pressure still grows with active sequence count times context length. Quantization therefore increases headroom and can improve hit rate and TTFT, but it does not eliminate the concurrency and tiering problem.

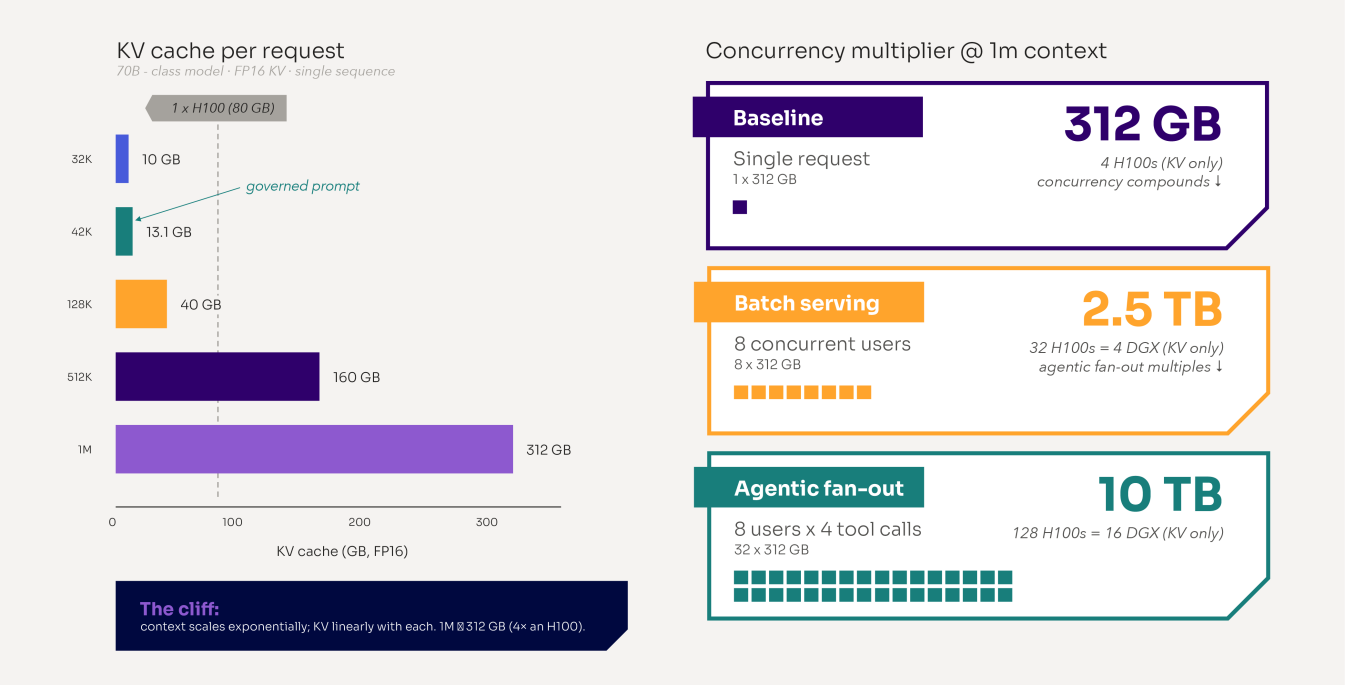


*Figure 6. The KV cache cliff. Note: Anchored to NVIDIA "Mastering LLM Techniques: Inference Optimization" (40GB @ 128K). 70B / 80-layer / GQA-8 / FP16. §6.1's ~42K example is a realistic baseline for enterprise assistants.*

### 6.4 Condensed concurrency walkthrough

Long-context sizing must account not only for prompt length, but for how many sequences are active at once. Concurrency multiplies KV footprint and determines whether state remains memory-resident or spills across tiers.

Define:

$U$ = active user sessions at peak, also known as concurrent users issuing requests

$A$ = average in-flight sequences per user objective including agentic fanout, retries, parallel tool calls, or speculative decoding

$N$ = total active sequences in the system

The relationship is:

$$N = U \times A$$

A system may report 120 active users ($U = 120$), but if a single objective triggers parallel steps, the effective sequence count rises.

If:

$$U = 120, \qquad A = 1.3$$

then:

$$N = 156$$

KV footprint scales with $N$, not with user count alone.

Across the fleet:

$$KV_{\text{fleet}} \approx N \times KV_{\text{per sequence}}$$

If a compiled prompt implies ~13.1 GB of KV per active sequence as shown in §0, then:

$$KV_{\text{fleet}} \approx 156 \times 13.1\ \text{GB} \approx 2\ \text{TB}$$

2TB of in-flight KV is not a corner case, it is a predictable outcome of long context combined with moderate concurrency.

$A$ is not a model constant, it is an operational average. It captures how many sequences your serving stack keeps in flight per user objective, typically >1 when agents fan out into parallel steps (retrieval + tool calls), retries, or speculative decoding.

Agentic frameworks like ReAct establish how a single user objective generates a multi-step trajectory of reasoning and tool invocations, each separately materializing KV state.[29] Preble adds the parallelism link directly, describing programming workflows in which the model may be invoked several times in parallel to generate multiple candidate programs and then select among them.[13] On that basis, using $A = 1.3$ as a conservative illustrative value is reasonable for lightly agentic systems, while we note that materially higher values are plausible for retry-heavy, candidate-generation, or branch-parallel designs.

Operators should compute $U$ and $A$ from telemetry, with $A$ estimated as **active sequences ÷ active user sessions,** and size tiers against p95/p99 TTFT and acceptable ITL jitter under peak $N$, not average load.

# 7. A practical storage hierarchy for inference state

A practical hierarchy distinguishes recomputable but latency-sensitive inference context (KV) from durable data such as documents, artifacts, long-retention logs. Mooncake, the serving platform for Moonshot AI's Kimi, reports this architecture in production, using hierarchical KV cache management and prefetching to reduce decode-path stalls.[30] NVIDIA describes Context Memory eXtensions (CMX) as a pod-level flash-based context memory tier intended to bridge GPU memory and general shared storage for large-scale inference.[31,32]

For procurement clarity, specify baseline interface and form factor constraints early, for example NVMe/PCIe and modern data center form factors such as EDSFF.[33,34]

Where storage is part of the KV path for instance in-paging, restore, or spill, reduce CPU overhead and jitter where feasible. For example, GPUDirect Storage documents a direct DMA data path between storage and GPU memory that avoids a CPU bounce buffer. Note that platform support varies.[35]

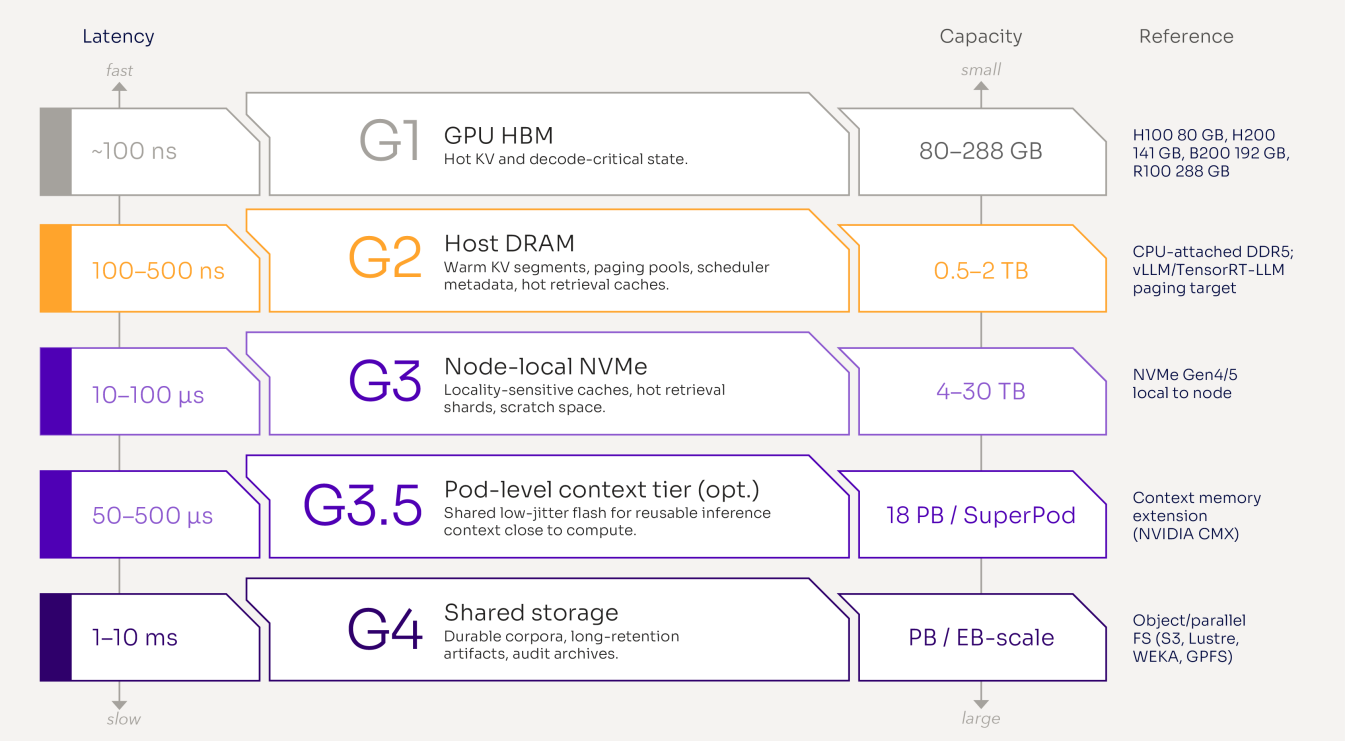


*Figure 7. Inference storage hierarchy. Note: Representative ranges for NVIDIA H100/H200/B200/R100 inference nodes. Latencies are order-of-magnitude; actual values depend on interconnect, page size, and software stack. G3.5 ranges are vendor-stated and should be treated as directional.*[31,32]

G3.5 exists as a distinct architectural tier because G1 and G2 cannot serve as the residency floor for reusable inference context. Under production pressure (concurrent sessions, agentic fan-out, long-context prefill) HBM and DRAM behave as ephemeral working windows rather than durable homes; registered context is eagerly evicted downward as new work arrives. G3 is a node-local tier, and in pod-scale disaggregated inference, KV state must occupy a shared resource. G3.5 is the architectural response: an Ethernet-attached flash tier with longer effective residency than G1/G2, sized to absorb that eviction pressure while keeping the durable G4 tier off the latency-critical path.

As implemented today it is best understood as a vendor-defined architectural concept, not yet a benchmark-validated industry primitive. The NVIDIA BlueField-4 CMX blog defines it as an Ethernet-attached flash tier positioned between local memory and durable storage, with an operating model in which KV blocks are prestaged into higher-speed memory before decode begins. NVIDIA claims up to 5x higher tokens-per-second and 5x better power efficiency versus traditional storage approaches using best-case vendor projections.[31] These are vendor-stated targets, not independent benchmark results, so treat them as directional until validated by third-party measurement.

Because eager eviction makes restore the dominant G3.5 access pattern at steady state, end-to-end restore latency from NVMe back into active GPU-resident serving state is the qualification metric that matters. Direct public measurements remain limited. Two data points help bracket the problem.

- SGLang's HiCache design implements hierarchical KV caching with configurable prefetch termination policies: best_effort, wait_complete, and timeout. This demonstrates that restore is latency-sensitive enough to require runtime policy control.[36] The existence of a timeout policy implies that restore can be slow enough to compete with recomputation which is exactly the tradeoff operators must measure.

- Oracle's NVMe latency measurements show raw command latencies in the single-digit to low-tens-of-microsecond range.[37] This establishes a floor. Media latency alone is fast, but end-to-end restore includes software overhead, batching, DMA setup, and cross-tier memory movement. The gap between raw media latency and delivered restore latency is where engineering effort, as well as qualification testing, should focus.

### 7.1 Tier placement rule: Read-dominant vs. mixed/write-influenced workloads

Tier placement for inference is a hierarchy decision before it is a media decision. The hierarchy above (G1–G4) determines where a workload class lives based on its access pattern, residency requirements, and tail-latency budget. Within a chosen tier, the read-vs-write character of the workload then guides media qualification.

A practical default within G3 and G3.5 distinguishes read-dominant from mixed/write-influenced placements:

- Read-dominant placements — chunk stores, warm corpus retention, large embedding indexes where writes are bounded and managed — can be served by high-density capacity media. The dominant access pattern is the one that needs latency assurance, and read paths benefit from the capacity headroom that density-optimized media provides.

- Mixed or write-influenced placements — tool outputs, scratch/build caches, continuous artifact logging — need media qualified for sustained mixed-load behavior. Write churn at the device layer can poison read tail latency on the same media when isolation is incomplete.

Both placements should be validated. Run workload-representative mixed-load p99/p99.9 tests in your environment.[6,7] The rule above is a deployment hypothesis, not a device guarantee.

Public vendor guidance commonly positions TLC as a good fit for mixed and write workloads, while QLC aligns best with large-scale, read-centric data needs where writes are bounded and managed.[26,38]

Density note: High-capacity QLC eSSDs are already shipping at ~122TB-class capacities, and the industry roadmap now includes ~245.76TB to 256TB-class devices and demonstrations. These density milestones substantially expand feasible per-rack capacity for read-dominant inference datasets, but they do not replace workload-specific QoS qualification. Pay attention to tail latency under concurrency and mixed load.[6,7,39,40,41]

### 7.2 Advanced (optional) isolation levers: ZNS and FDP

Treat ZNS and FDP as advanced levers, not defaults. Both require device capability and software-stack support where benefits are workload-dependent and must be validated with mixed-load tail-latency tests. Evaluate them when standard namespace/device isolation (§8.1) is insufficient and write-amplification or GC interference remains a measured problem.

The core mechanism is the same in both cases: The host tells the SSD which data has similar lifetimes, so the device can co-locate it and avoid mixing short-lived writes with long-lived data during garbage collection. NVMe Zoned Namespaces (ZNS) enforces this through sequential-write zones;[42] Flexible Data Placement (FDP) achieves it within standard NVMe by tagging writes with placement hints that map to separate reclaim units.[43,44]

Practitioner evidence supports the mechanism. ZenFS (RocksDB on ZNS) uses write-lifetime hints to co-locate data by expected lifespan, reducing write amplification and avoiding background GC in both the file system and device layers.[45] Samsung's FDP guidance describes separating data lifetimes into different reclaim units to reduce write amplification and improve reclamation efficiency.[46] For test methodology, zonedstorage.io documents fio configurations that emulate parallelism on ZNS devices,[47] providing a practical starting point for workload-representative qualification.

# 8. Procurement-grade qualification checklist

Procurement must shift from peak metrics to tail QoS. Do not qualify inference-and-RAG storage from a single high-QD peak IOPS result. Require p99 and p99.9 latency at QD=1–4 under target concurrency and mixed read/write conditions.[6] Run long enough to observe background effects like cache transitions, internal maintenance, and garbage collection.[7] Capture latency distributions (e.g., histograms), not just averages, and correlate tail excursions with workload phases such as retrieval bursts, and tool artifact writes.

### 8.1 Five procurement-grade requirements for inference storage

If you can only specify five requirements, these are recommended:

| Requirement | What to ask for | Why it matters |
|---|---|---|
| Low-QD concurrent latency | p99 latency at QD1–4 under 128+ concurrent jobs (many submission queues), not a single job at high QD. | Emulates the actual per-thread retrieval behavior of RAG under production concurrency (§0). |
| Mixed R/W tail latency | p99 at a representative read/write mix (e.g., 95/5) after steady state, including background maintenance behavior. | Exposes garbage-collection-induced tail spikes that poison retrieval latency when reads and writes share a device (§0). |
| Full latency distribution | Latency histograms (p50/p95/p99/p99.9), not only averages or peak IOPS. | Averages hide the tail events that dominate user-visible TTFT under fanout.[5] |
| Steady-state duration | Results measured after 30–60 minutes per point (or longer for sustained workloads). | Short runs capture burst cache behavior, not the steady-state regime where maintenance effects appear.[7] |
| Write-plane isolation | Evidence that artifact-write planes are isolated from latency-critical retrieval reads (separate devices/namespaces or enforced QoS arbitration). | Prevents tool-output churn from Joe's CI pipeline (§0) from spiking his retrieval p99. |

*Table 3. Procurement-grade requirements.*

### 8.2 Example test matrix

| Dimension | Example values | Why it matters |
|---|---|---|
| Block sizes | 4 KiB; 16 KiB; 64 KiB; 256 KiB | Cover metadata and chunk-payload regimes (packing changes the mix). |
| Per-job queue depth | 1; 2; 4 | Emulate per-thread retrieval behavior; avoid sizing from a single QD=32 run. |
| Number of concurrent jobs | 64; 128; 256 (scale to target QPS) | Emulate many sessions and submission queues. |
| Read/write mix | 100/0; 95/5; 80/20 | Expose mixed-load interference and maintenance tails.[7] |
| Steady-state duration | 30–60 minutes per point | Allow background maintenance and cache effects to stabilize. |
| Metrics | p50/p95/p99/p99.9; throughput; CPU utilization | Tail behavior dominates SLOs; orchestration overhead matters.[6] |

*Table 4. Example test matrix. Note: values are starting points and should be tuned to observed block-size distributions and concurrency.*

# 9. Operational and security implications

### 9.1 Prompt-aware telemetry (minimum viable set)

The telemetry below closes the loop between user-visible SLOs and infrastructure causes. When TTFT p99 moves, this instrumentation should distinguish whether the driver is retrieval stragglers, prefix restore jitter, KV eviction/restore, or tool/artifact interference. Each implies a different tiering and isolation action.

- Token breakdown by component per request/objective (e.g., policy, retrieval, tools, history).
- TTFT split: Queueing, retrieval/tool assembly, prefill, first-token decode matching the five-term decomposition in §0.
- Cache metrics (e.g., cached tokens, hit rates, retention behavior).[8]
- KV metrics: Estimated KV GB in flight; eviction rate; spill/restore rate; recompute rate.[2,4]
- Storage metrics: p99 latency under mixed load; noisy-neighbor indicators; shard-level straggler counts.[5]

### 9.2 Tool-connected assistants expand the attack surface

Tool protocols make assistants more useful, but also expand the attack surface. MCP security guidance and OWASP's LLM application risk list highlight risks such as prompt injection, excessive agency, and confused-deputy failure modes.[12,48]

In production deployments, security and governance controls often increase state. Policy context and tool allow lists can enlarge stable prefixes. Redaction and sandboxing can add preprocessing steps and audit requirements can turn tool outputs into retained artifacts. These controls increase both token footprint, i.e., prefill/KV pressure, and write-plane churn (logs, traces, and evidence bundles.

For Joe's deployment as shown in §0, this means every tool invocation generates an audit record, every code diff is retained for compliance, and every CI test result is logged. These writes share the storage fabric with his retrieval reads. Without isolation, Joe's own audit trail poisons his own retrieval p99.

Treat security posture as an explicit sizing input. Measure incremental tokens and incremental artifact write volume introduced by each control, then isolate the write plane so it cannot poison retrieval p99 under load.

# 10. Conclusion

Table 1 is a hypothesis generator. Each cell predicts a testable I/O behavior. The procurement checklist in §8 is a test plan. The telemetry in §9 closes the loop.

The chain is: 1) prompt anatomy predicts state anatomy, 2) state anatomy predicts I/O behavior, and 3) I/O behavior predicts user-visible latency. Two implications follow for procurement.

Long context and agentic workflows turn "GB per session" into "TB per fleet" as KV scales with token length and concurrency. Controlling this requires a deliberate memory/storage hierarchy with measured policies for reuse, eviction, and restore.

RAG behaves like a low-queue-depth random-read workload under concurrency, gated by p99/p99.9 latency across the assembly path, not by peak IOPS at high queue depth.

The next step for any deployment is to run the qualification matrix, instrument the telemetry, and validate the predictions. Where they fail, the mapping needs refinement. Where they hold, you have a procurement specification you can defend. Storage belongs in the day-zero design conversation for AI serving stacks, alongside compute, memory, and network — not as a day-two retrofit.

**Appendix A.**

# KV cache approximate sizing cheat sheet

A generic bytes-per-token estimate for transformer decoders:

$$KV_{\text{bytes/token}} \approx 2 \times L \times H_{kv} \times D \times \text{bytes/element}$$

where $L$ is the number of layers, $H_{kv}$ is the number of key/value heads, $D$ is the head dimension, and $\text{bytes/element}$ is set by the KV cache representation (precision and any quantization), which is implementation-dependent. The factor 2 accounts for keys and values.

The KV footprint scales approximately as:

$$KV_{\text{total}} \approx KV_{\text{bytes/token}} \times \text{tokens in context} \times \text{active sequences} \; (+ \text{allocator overhead})$$

Grouped-query attention and multi-query attention can reduce $H_{kv}$; KV quantization reduces $\text{bytes/element}$. Use this formula as a starting point and validate with serving-stack telemetry.[4,28]

## Appendix B.

# RAG I/O envelope by architecture (illustrative)

The effective I/O envelope depends on where the index resides and how chunks are packed. Use this table to turn architecture choices into testable I/O profiles.

Cross-reference: §5 describes the low-QD random-read storm model that these architectures produce; §0 maps workload characteristics to placement tiers.

| RAG architecture pattern | Index placement | Chunk placement | Dominant read sizes (illustrative) | Write-amplification risks (ingest) | Best-fit NAND media (default) |
|---|---|---|---|---|---|
| **HNSW or IVF-PQ in DRAM; chunks on NVMe** | DRAM | Local NVMe | Chunk fetch ~64 KiB–1 MiB; metadata ~4–16 KiB | Chunk updates + compaction, if co-located with serving reads | QLC for read-dominant chunks; TLC for build plane (isolate).[26,38] |
| **Memory-mapped index on NVMe; chunks on NVMe** | NVMe (mmap) | NVMe | Mixed: small postings reads + chunk reads; higher small-read rate | Index rebuild + compaction can create mixed load | Prefer TLC or high-QoS QLC with strict isolation.[26] |
| **Hybrid BM25 + vector (separate postings and embeddings)** | DRAM + NVMe | NVMe | Mixed: small postings reads + chunk reads | Posting list updates; embedding refresh cycles | TLC for write-heavy postings; QLC for cold corpora (isolate).[26] |

*Table 5. RAG I/O envelope by architecture pattern.*

## Appendix C.

# Benchmarking checklist for RAG reality

- Use many concurrent jobs at QD=1–4 to emulate independent sessions; avoid sizing from a single high-QD run.
- Measure p99 and p99.9 under mixed load with background writes representative of artifact churn.[7]
- Run long enough to reach steady state (minutes to hours) to capture internal maintenance effects.[7]
- Collect device-level latency telemetry and correlate tail excursions with workload phases (retrieval bursts, tool artifact writes, index rebuild).

## Appendix D.

# Glossary

| Term | Definition |
|---|---|
| TTFT | Time to first token. Elapsed time from request submission to first generated token. |
| ITL | Inter-token latency. Per-token time during decode; user-visible as “typing” smoothness or jitter. |
| KV cache | Key/value tensors stored for previously processed tokens to avoid recomputation during attention.[3] |
| Prefill | Phase that processes the full input prompt and materializes KV for prompt tokens. |
| Decode | Phase that generates output tokens iteratively, reading prior KV and appending new KV. |
| RAG | Retrieval-augmented generation: retrieval of external evidence (index + chunk store) to ground responses.[10] |
| Tail latency | High-percentile latency (p99/p99.9) that often dominates end-to-end SLOs under fanout.[5] |

*Table 6. Glossary.*

# References

1. Google, “Long context,” Gemini API documentation. Available: https://ai.google.dev/gemini-api/docs/long-context (accessed Feb. 24, 2026).
2. NVIDIA, “Accelerate Large-Scale LLM Inference and KV Cache Offload with CPU-GPU Memory Sharing,” NVIDIA Technical Blog, Sep. 5, 2025. Available: https://developer.nvidia.com/blog/accelerate-large-scale-llm-inference-and-kv-cache-offload-with-cpu-gpu-memory-sharing/ (accessed Feb. 24, 2026).
3. Hugging Face, “KV cache strategies,” Transformers documentation (v4.55.4). Available: https://huggingface.co/docs/transformers/v4.55.4/en/kv_cache (accessed Feb. 24, 2026).
4. W. Kwon et al., “Efficient Memory Management for Large Language Model Serving with PagedAttention,” arXiv:2309.06180. Available: https://arxiv.org/abs/2309.06180 (accessed Feb. 24, 2026).
5. J. Dean and L. A. Barroso, “The Tail at Scale,” Communications of the ACM, vol. 56, pp. 74–80, 2013. Available: https://research.google/pubs/the-tail-at-scale/ (accessed Feb. 24, 2026).
6. Intel, “Performance Benchmarking for PCIe and NVMe Enterprise Solid-State Drives,” white paper, Feb. 2015. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/performance-pcie-nvme-enterprise-ssds-white-paper.pdf (accessed Feb. 24, 2026).
7. S.-H. Cho et al., “Efficient Garbage Collection Algorithm for Low Latency SSD,” Electronics, vol. 11, no. 7, 1084, 2022. Available: https://www.mdpi.com/2079-9292/11/7/1084 (accessed Feb. 24, 2026).
8. OpenAI, “Prompt caching,” OpenAI API documentation. Available: https://developers.openai.com/api/docs/guides/prompt-caching/ (accessed Feb. 24, 2026).
9. WEKA, “Why Prefill has Become the Bottleneck in Inference—and How Augmented Memory Grid Helps,” WEKA Blog, June 6, 2025. Available: https://www.weka.io/blog/ai-ml/why-prefill-has-become-the-bottleneck-in-inference-and-how-augmented-memory-grid-helps/ (accessed Feb. 24, 2026).
10. P. Lewis et al., “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” arXiv:2005.11401. Available: https://arxiv.org/abs/2005.11401 (accessed Feb. 24, 2026).
11. Anthropic, “Code execution with MCP: Building more efficient agents,” Engineering at Anthropic, Nov. 4, 2025. Available: https://www.anthropic.com/engineering/code-execution-with-mcp (accessed Feb. 24, 2026).
12. Model Context Protocol, “Security Best Practices,” Model Context Protocol documentation. Available: https://modelcontextprotocol.io/docs/tutorials/security/security_best_practices (accessed Feb. 24, 2026).
13. V. Srivatsa, Z. He, R. Abhyankar, D. Li, and Y. Zhang, “Preble: Efficient Distributed Prompt Scheduling for LLM Serving,” arXiv:2407.00023. Available: https://arxiv.org/abs/2407.00023 (accessed Mar. 9, 2026).
14. The SGLang Team, “SGLang v0.4: Zero-Overhead Batch Scheduler, Cache-Aware Load Balancer, Faster Structured Outputs,” LMSYS Blog, Dec. 4, 2024. Available: https://lmsys.org/blog/2024-12-04-sglang-v0-4/ (accessed Mar. 9, 2026).

15. vLLM Project, “vLLM v0.6.0: 2.7x Throughput Improvement and 5x Latency Reduction,” vLLM Blog, Sep. 5, 2024. Available: https://blog.vllm.ai/2024/09/05/perf-update.html (accessed Mar. 9, 2026).

16. NVIDIA, “Metrics — NVIDIA NIM LLMs Benchmarking,” NVIDIA documentation. Available: https://docs.nvidia.com/nim/benchmarking/llm/latest/metrics.html (accessed Mar. 9, 2026).

17. V. Paruthi, “How Input Token Count Impacts the Latency of AI Chat Tools,” Glean Engineering Blog. Available: https://www.glean.com/blog/glean-input-token-llm-latency (accessed Mar. 9, 2026).

18. Y. Zhong et al., “DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving,” in Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’24), 2024. Available: https://arxiv.org/pdf/2401.09670 (accessed Mar. 9, 2026).

19. Databricks, “LLM Inference Performance Engineering: Best Practices,” Databricks Blog. Available: https://www.databricks.com/blog/llm-inference-performance-engineering-best-practices (accessed Mar. 9, 2026).

20. vLLM Project, “Metrics — vLLM v0.8.2 Design Documentation,” vLLM documentation. Available: https://docs.vllm.ai/en/v0.8.2/design/v1/metrics.html (accessed Mar. 9, 2026).

21. M. Shen, M. Umar, K. Maeng, G. E. Suh, and U. Gupta, “Towards Understanding Systems Trade-offs in Retrieval-Augmented Generation Model Inference,” arXiv:2412.11854. Available: https://arxiv.org/abs/2412.11854 (accessed Mar. 9, 2026).

22. Z. Ren, K. Doekemeijer, N. Tehrany, and A. Trivedi, “BFQ, Multiqueue-Deadline, or Kyber? Performance Characterization of Linux Storage Schedulers in the NVMe Era,” in Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE ’24), 2024. Available: https://research.spec.org/icpe_proceedings/2024/proceedings/p154.pdf (accessed Mar. 9, 2026).

23. NVM Express, “NVMe over Fabrics (oF) Specification,” specification landing page, NVMe-oF 1.1a (historical reference; no further development planned). Available: https://nvmexpress.org/specification/nvme-of-specification/ (accessed Feb. 24, 2026).

24. Anthropic, “Introducing advanced tool use on the Claude Developer Platform,” Engineering at Anthropic, Nov. 24, 2025. Available: https://www.anthropic.com/engineering/advanced-tool-use (accessed Mar. 9, 2026).

25. OpenAI, “Function calling,” OpenAI API documentation. Available: https://platform.openai.com/docs/guides/function-calling (accessed Mar. 9, 2026).

26. Solidigm, “QLC NVMe SSDs Are Optimal for Modern Workloads,” solution brief (workload guide), 2023. Available: https://www.solidigm.com/content/dam/solidigm/en/site/products/technology/qlc-nand-ssds-are-optimal/documents/qlc-nand-workload-guide.pdf (accessed Feb. 24, 2026).

27. NVIDIA, “FP8 Quantization,” TensorRT-LLM documentation. Available: https://nvidia.github.io/TensorRT-LLM/performance/performance-tuning-guide/fp8-quantization.html (accessed Mar. 9, 2026).

28. NVIDIA, “Optimizing Inference for Long Context and Large Batch Sizes with NVFP4 KV Cache,” NVIDIA Technical Blog, Dec. 8, 2025. Available: https://developer.nvidia.com/blog/optimizing-inference-for-long-context-and-large-batch-sizes-with-nvfp4-kv-cache/ (accessed Feb. 24, 2026).

29. S. Yao et al., “ReAct: Synergizing Reasoning and Acting in Language Models,” arXiv:2210.03629. Available: https://arxiv.org/abs/2210.03629 (accessed Feb. 24, 2026).

30. R. Qin et al., “Mooncake: A KVCache-centric Disaggregated Architecture for LLM Serving,” arXiv:2407.00079. Available: https://arxiv.org/abs/2407.00079 (accessed Mar. 9, 2026).

31. NVIDIA, “Introducing NVIDIA BlueField-4-Powered Inference Context Memory Storage Platform for the Next Frontier of AI,” NVIDIA Technical Blog, Jan. 6, 2026. Available: https://developer.nvidia.com/blog/introducing-nvidia-bluefield-4-powered-inference-context-memory-storage-platform-for-the-next-frontier-of-ai/ (accessed Feb. 24, 2026).

32. NVIDIA, “NVIDIA BlueField-4 Powers New Class of AI-Native Storage Infrastructure for the Next Frontier of AI,” NVIDIA Newsroom press release, Jan. 5, 2026. Available: https://nvidianews.nvidia.com/news/nvidia-bluefield-4-powers-new-class-of-ai-native-storage-infrastructure-for-the-next-frontier-of-ai (accessed Feb. 24, 2026).

33. NVM Express, “NVM Express Base Specification,” specification landing page, Base Spec Rev. 2.3, ratified Aug. 1, 2025. Available: https://nvmexpress.org/specification/nvm-express-base-specification/ (accessed Feb. 24, 2026).

34. SNIA, “SSD Form Factors,” SNIA technical page. Available: https://www.snia.org/node/14267 (accessed Feb. 24, 2026).

35. NVIDIA, “GPUDirect Storage,” NVIDIA documentation. Available: https://docs.nvidia.com/gpudirect-storage/ (accessed Feb. 24, 2026).

36. SGLang Team, “HiCache System Design and Optimization,” SGLang documentation. Available: https://docs.sglang.ai/advanced_features/hicache_design.html (accessed Mar. 9, 2026).

37. R. Shanmugavelu, “Measuring NVMe Latency,” Oracle Linux Blog, June 13, 2023. Available: https://blogs.oracle.com/linux/measuring-nvme-latency (accessed Mar. 9, 2026).

38. Micron, “Micron QLC NAND Technology,” product flyer, Rev. A, May 2018. Available: https://www.micron.com/content/dam/micron/global/public/products/product-flyer/qlc-technology-flyer.pdf (accessed Feb. 24, 2026).

39. Solidigm, “D5-P5336 High-Density PCIe 4.0 SSD,” product page. Available: https://www.solidigm.com/products/data-center/d5/p5336.html (accessed Feb. 24, 2026).

40. KIOXIA America, Inc., “KIOXIA Announces Industry’s First 245.76 TB NVMe SSD Built for the Demands of Generative AI Environments,” news release, July 21, 2025. Available: https://americas.kioxia.com/en-us/business/news/2025/ssd-20250721-1.html (accessed Feb. 24, 2026).

41. Sandisk, “Sandisk Showcases UltraQLC Technology Platform with Milestone Enterprise SSD Capacity at FMS 2025,” press release, Aug. 5, 2025. Available: https://www.sandisk.com/company/newsroom/press-releases/2025/2025-08-05-sandisk-showcases-ultraqlc-technology-platform-with-milestone-enterprise-ssd-capacity-at-fms-2025 (accessed Feb. 24, 2026).

42. NVM Express, “NVMe Zoned Namespaces (ZNS) Command Set Specification,” specification landing page, Rev. 1.4, ratified Aug. 1, 2025. Available: https://nvmexpress.org/specification/nvme-zoned-namespaces-zns-command-set-specification/ (accessed Feb. 24, 2026).

43. C. Sabol and R. Stenfort, “Hyperscale Innovation: Flexible Data Placement Mode (FDP),” NVM Express sponsored white paper. Available: https://nvmexpress.org/wp-content/uploads/Hyperscale-Innovation-Flexible-Data-Placement-Mode-FDP.pdf (accessed Feb. 24, 2026).

44. J. Edge, “Flexible data placement,” LWN.net, May 2, 2025. Available: https://lwn.net/Articles/1018642/ (accessed Mar. 9, 2026).

45. Zoned Storage, “RocksDB with ZenFS,” Zoned Storage documentation. Available: https://zonedstorage.io/docs/applications/zenfs (accessed Mar. 9, 2026).

46. Samsung, “Getting Started with Flexible Data Placement (FDP),” Samsung Semiconductor white paper. Available: https://download.semiconductor.samsung.com/resources/white-paper/getting-started-with-fdp-v4.pdf (accessed Mar. 9, 2026).

47. Zoned Storage, “fio Examples for NVMe ZNS Devices,” Zoned Storage documentation. Available: https://zonedstorage.io/docs/benchmarking/fio/zns-fio (accessed Mar. 9, 2026).

48. OWASP, “OWASP Top 10 for Large Language Model Applications (version 1.1),” OWASP Foundation. Available: https://owasp.org/www-project-top-10-for-large-language-model-applications/ (accessed Feb. 24, 2026).

# Legal Notices

All product plans, roadmaps, specifications, and product descriptions are subject to change without notice.
Nothing herein is intended to create any express or implied warranty, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, or any warranty arising from course of performance, course of dealing, or usage in trade.

The products described in this document may contain design defects or errors known as “errata,” which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your Solidigm representative or your distributor to obtain the latest specifications before placing your product order. For copies of this document, documents that are referenced within, or other Solidigm literature, please contact your Solidigm representative.

All products, computer systems, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice.

SOLIDIGM and the Solidigm “S” logo are trademarks of Solidigm Inc., registered in the United States, People’s Republic of China, Taiwan, Hong Kong, Singapore, the European Union, the United Kingdom, Mexico, and other countries.

Solidigm may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Solidigm reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase.

Performance results are based on testing as of dates shown in the configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.

Solidigm or Intel optimizations, for Solidigm or Intel compilers or other products, may not provide optimized performance to the same degree for non-Solidigm or Intel products. Solidigm or Intel technologies may require enabled hardware, software, or service activation.

Your costs and results may vary.

Solidigm does not control or audit third-party data. You should consult other sources to evaluate accuracy.
Some results have been estimated or simulated using internal Solidigm analysis or architecture simulation or modeling, and provided to you for information purposes only. Any differences in your system hardware, software or configuration may affect your actual performance.