# LatentStore: An Efficient Latent-First Storage System for AI-Generated Images

Anonymous Authors, Paper #258

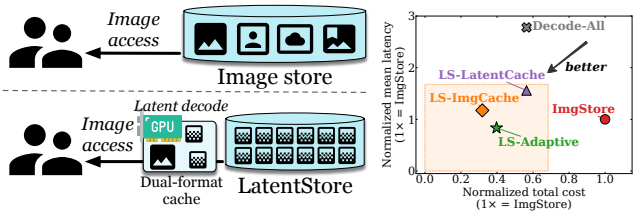
## Abstract

The explosive growth of AI-generated images has created a sustainability challenge for storage infrastructure. Platforms like Midjourney and Firefly already host billions of generative images, yet conventional object stores persist them as blobs with full-resolution pixels, consuming huge amounts of storage capacity and bandwidth. Unlike natural photos, however, AI-generated images can be deterministically reconstructed from compact, model-native latent tensors, making persistent image storage fundamentally redundant.

This paper presents LatentStore, a latent-first storage system for AI-generated images. LatentStore treats compressed latents as durable storage objects and uses on-demand GPU reconstruction on the read path to trade inexpensive compute for large persistent storage savings. Our design is guided by the first large-scale analysis of AI-generated image access we are aware of, based on a 35-month, 2-billion-request production trace from a major generative-content platform. Motivated by the trace analysis, LatentStore keeps frequently accessed images in decoded pixel format for fast hits, stores less-active objects as compressed latents to expand effective cache capacity, and continuously adjusts the splits between the image and latent cache to optimize user-perceived access latency. We build a LatentStore prototype and evaluate it with the production trace. LatentStore reduces persistent storage by 78.7% with competitive or even lower mean and tail latency over a pure image-based storage.

## 1 Introduction

The proliferation of text-to-image generative models has created an unprecedented volume of synthetic visual content. Platforms such as Midjourney [2] now produce over 34 million images per day [21], while Adobe Firefly has generated more than 29 billion images since its 2023 launch [3]. At typical resolutions (1024×1024), storing 100 billion losslessly compressed PNGs requires roughly 150 PB, with costs scaling linearly as new images are created every day, making storage a dominant and fast-growing infrastructure expense for generative-content platforms.




**(a)** Conventional image store vs. LatentStore. **(b)** Cost–latency tradeoff.

**Figure 1:** Illustration of latent-first storage. (a) Conventional object stores persist AI-generated images as opaque blobs, whereas LatentStore (LS) stores compact model-native latents (intermediate state) and reconstructs images on demand. (b) Cost–latency tradeoff of five storage strategies. LatentStore achieves low cost and latency.

Existing large-scale image storage systems, including Facebook's Haystack [10] and f4 [40], LinkedIn's Ambry [41], and Tencent's photo store [64], treat images as opaque blobs. Their optimization levers are limited to *where* to place blobs, *how* to replicate them, and *what* to cache. These systems cannot exploit a property unique to AI-generated content: every image is a *deterministic function of a compact, model-native latent tensor [52]*, an intermediate representation that can be decoded into pixel images, making persistent storage of full-resolution image redundant.

Modern diffusion models produce images through a two-stage pipeline: an expensive iterative diffusion process that takes seconds to generate a small latent tensor, followed by a lightweight VAE decoder [14, 20, 28, 51, 55] that maps the latent to images in tens of milliseconds. The decode is deterministic (the same latent always yields the same image) and the latent is roughly 5× smaller than the corresponding PNG. This asymmetry reveals the key insight of this paper: *Image generation is expensive but performed only once. Storing the compact latent minimizes persistent storage cost, while image reconstruction via GPU decode is cheap enough to be performed on demand without inflating user-perceived latency.*

A *latent-first* storage system can therefore replace stored images with much smaller latents and reconstruct on the fly. However, realizing the storage savings without sacrificing user-facing performance introduces new challenges: requests

that cannot be served from a decoded image cache now triggers a GPU decode rather than a simple storage read.

Analysis of a large production trace from a major generative-content platform CompanyX shows that image popularity is heavily skewed yet rapidly decaying, re-access intervals span seconds to months, and even a well-sized cache leaves a persistent miss residual. These properties make a latent-first design nontrivial. If the system caches decoded images, popular requests are fast because they avoid GPU decode, but the cache holds fewer objects and more requests fall through to slower paths. If the system caches only latents, the same cache budget covers more objects, but every cache hit still incurs GPU decode cost. The right choice is therefore not a fixed format: newly popular images may be worth keeping as decoded images, while colder images are better kept as compact latents. As image popularity changes over time and new images continuously arrives, the system must adaptively adjust how much cache space is devoted to different cache formats (images vs. latents).

We present LatentStore, a novel latent-first storage system that stores AI-generated images as compressed latents and reconstructs them on demand using GPU decoding. Fig. 1a shows the shift from image persistence to latent-first storage, and Fig. 1b previews the resulting cost–latency tradeoff.

This paper makes the following contributions:

- **Workload characterization:** To the best of our knowledge, we present the first large-scale analysis of how users access AI-generated images , using a 35-month, 2-billion-request production trace from a major generative-content platform.
- **Latent-first storage:** To our knowledge, LatentStore is the first storage system for AI-generated images that treats compressed, model-native latents as durable storage objects and uses on-demand GPU reconstruction on the read path to trade inexpensive compute for large storage savings.
- **Dual-format adaptive cache:** LatentStore makes latent-first storage practical through a dual-format cache that caches hot objects as decoded images for fast hits and colder objects as compact latents for coverage and an adaptive cache resizer that adjusts the split between above two parts.
- **End-to-end evaluation:** We build a LatentStore prototype and evaluate it using the production trace[1]. LatentStore reduces persistent storage by 78.7%, lowers mean and P99 read latency by 17% and 18% over PNG-only storage, and projects over 60% cumulative cost savings over 20 years.

# 2 Background

## 2.1 The Scale of AI-Generated Images

The proliferation of text-to-image generative models has created an unprecedented volume of synthetic visual content. As shown in Fig. 2, Adobe Firefly alone has exhibited explosive growth since its April 2023 launch: the platform reached 3 billion cumulative images within its first six months, crossed

[1] LatentStore code and trace will be open-sourced upon paper acceptance.

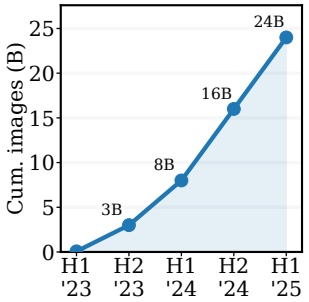



**Figure 2:** Adobe Firefly sees explosive gen-image growth [3].

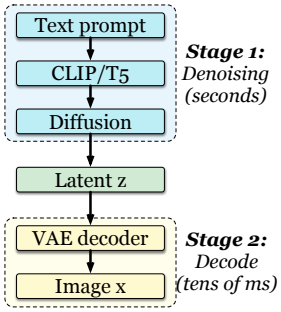



**Figure 3:** Two-stage text-to-image pipeline.

16 billion by the end of 2024, and surpassed 24 billion by mid-2025, with the monthly generation rate accelerating from roughly 500 million to over 2 billion images per month [3]. This trajectory is not unique to Adobe—Midjourney produces over 34 million images per day [21], and open-source platforms built around Stable Diffusion [19, 46, 52, 53] and FLUX [29] contribute further to the global supply of AI-generated imagery.

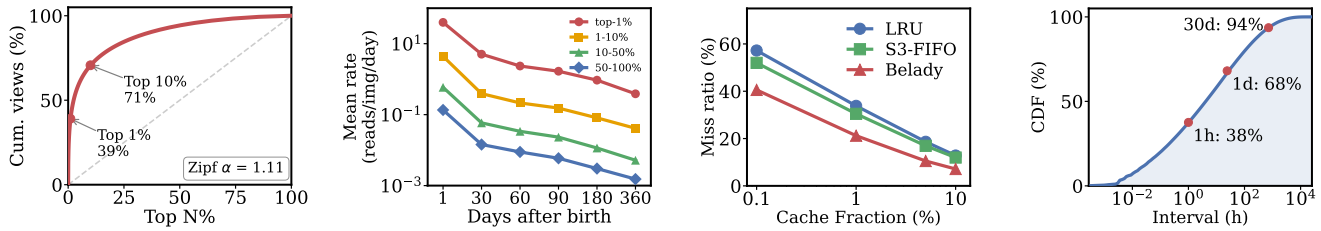
This rapid growth translates directly into a storage challenge. At typical resolutions ($1024 \times 1024$), storing 100 billion images as losslessly compressed PNGs requires on the order of 150 PB, with storage capacity costs scaling linearly with the ever-expanding working set. Unlike camera photos, which are irreproducible records of a physical scene, AI-generated images are *deterministic outputs of a computational process* fully characterized by a compact latent tensor, opening an optimization axis that conventional image storage systems cannot exploit.

## 2.2 Diffusion Models and VAE Decoding

Modern text-to-image models, including Stable Diffusion and FLUX, generate images through a two-stage VAE (variational autoencoder) pipeline [28]. See Fig. 3.

**Stage 1: Denoising.** A text prompt is first encoded into a conditioning vector by a language model such as CLIP [15, 47, 50] or T5 [48]. A diffusion process then iteratively denoises a random tensor in a *latent space*, guided by the conditioning signal, over 20–50 steps. The output is a latent tensor $\mathbf{z} \in \mathbb{R}^{c \times h \times w}$, where $c$ is the number of latent channels and $h \times w$ is the spatial resolution of the latent grid.

**Stage 2: VAE decoding.** The VAE decoder maps the latent tensor to the pixel space: $\mathbf{x} = \mathcal{D}(\mathbf{z})$, producing an RGB image $\mathbf{x} \in \mathbb{R}^{3 \times H \times W}$. The decoder is a *deterministic* feed-forward neural network with no sampling or stochastic components, so the same latent always yields a bit-identical pixel output on the same GPU and software stack. This single forward pass is orders of magnitude cheaper than the iterative diffusion process. Note that while decoding in LLM text generation is memory-bandwidth-bound [4], denoising and decoding in image generation are compute-bound—the denoising and decoding latency scales linearly with batch size.




**(a)** Popularity CDF. **(b)** Access decay vs. age. **(c)** Miss ratio vs. cache size. **(d)** Re-access interval CDF.

**Figure 4:** CompanyX trace characterization. (a) Image popularity CDF. (b) Mean access rate vs. age, stratified by lifetime-view quartile. (c) Miss ratio vs. cache size for three policies. (d) CDF of intervals between consecutive accesses.

# 3 Motivation

This section motivates the design of LatentStore by analyzing a production trace (§3.1) and characterizing the cost of on-demand pixel reconstruction (§3.2).

## 3.1 Production Trace Analysis

To understand the access characteristics of AI-generated images at scale, we collected a 35-month production access trace from CompanyX, one of the largest open platforms for generative-AI imagery, where creators publish and share model-generated images and community members browse, download, and remix them. The trace records over 2 billion requests from April 2023 to March 2026 and each request log has a *timestamp, image ID, model ID, and model version ID*. Table 1a summarizes the trace.

### 3.1.1 Workload Characterization

The dataset spans 1,049 days and records 2.07 billion anonymized image-view events across 92.3 million unique images generated by more than 710 K distinct models. The monthly request volume ranges from 45–70 million views, with an average of roughly 2 million requests per day. We make four observations that motivated LatentStore.

**Observation #1 (O1): Skewed popularity, with a heavy tail.** Image popularity follows a Zipf-like distribution with $\alpha \approx 1.11$ (Fig. 4a). The top 1% of images account for 39% of all views, the top 10% account for 71%, while 69% of images receive fewer than ten views across the entire trace, and 15% are accessed only once.

***Design implication:*** This skew suggests that most generated images are rarely accessed after creation, making it inefficient to keep them in hot storage. Furthermore, because the images can be reproduced by the model, cold images could be regenerated on demand as long as the model remains available.

**Observation #2 (O2): Rapid post-birth decay, even for popular images.** Fig. 4b shows the mean per-image access rate as a function of days since first appearance, stratified by lifetime popularity. Even the most popular 1% of images see their access rate drop by over 100× within a year. All popularity tiers follow the same pattern: being "hot" is a transient *phase*, not a permanent property. Images go from newly uploaded to briefly popular and then quickly become cold, regardless of their eventual total view count.

***Design implication:*** Because popularity is a transient phase rather than an intrinsic property, the workload mix that a serving system faces changes continuously. Any cache format or cache sizing policy must therefore adapt online: a static configuration becomes increasingly mismatched as content ages and new images arrive.

**Observation #3 (O3): Non-trivial miss ratios persist even with caching.** Fig. 4c shows the miss ratio for LRU, S3-FIFO [61], and offline-optimal Belady [11] as the cache grows from 0.1% to 10% of the working set. Even at 10%, the best online policy (S3-FIFO) still misses roughly 12% of requests. A significant fraction of requests will therefore always fall through to the slow path.

***Design implication:*** We should size the cache carefully to strike a balance between cost and latency.

**Observation #4 (O4): Large variance in re-access intervals.** Fig. 4d shows the CDF of intervals between consecutive accesses to the same image, aggregated over the ∼78.1 M images that were accessed more than once. Roughly 38% of re-accesses happen within an hour and 68% within a day; the remaining 32% are spread across days, weeks, and months, with 6% beyond 30 days. This explains why we still observe over 10% miss ratio at a large cache size.

***Design implication:*** The large variance in re-access intervals indicates that a multi-tiered cache is needed to reduce both cost and latency.

## 3.2 Diffusion Model Compute Characteristics

The two-stage computation in the diffusion model presents a unique opportunity to improve storage efficiency—storing the intermediate state (latent) instead of the raw image and reconstructing the images on demand.

This subsection analyzes the compute characteristics of diffusion models and show that (1) latent is 4–6× smaller than the corresponding image across different models; (2) decode is around two orders of magnitude faster than generation.

**Latents are smaller than images.** Table 1b compares the decoder parameter count, raw latent shape and size, compressed latent size, and the corresponding pixel and PNG image sizes across three widely used model families at 1024×1024 resolution. Across all models, the raw FP16 latent is 128–512 KB depending on the number of latent channels, while uncompressed pixels occupy 3.0 MB and PNGs 1.3–1.4 MB. For SD 3.5 at 1024×1024, the raw latent tensor is roughly 6×

**Table 1:** (a) Summary of the 35-months production trace from CompanyX. (b) Decoder, latent, and image size across model families at 1024×1024. Pixel is uncompressed RGB; Lat Compressed is pcodec-compressed latent; PNG is the average file size from our evaluation dataset. SD 1.5 targets 512×512 and does not support 1024×1024 generation. (c) Per-image latency for SD 3.5 on different NVIDIA GPUs. Image size 1024 × 1024, 28 denoising steps, FP16, batch size 1.

**(a)** Trace summary.

| Metric | Value |
|---|---|
| Requests | 2.07 B |
| Images | 92.3 M |
| Models | 710 K |

**(b)** Decoder & latent size.

| | | Image | | Latent | | |
|---|---|---|---|---|---|---|
| Model | Params | Pixel | PNG | Shape | Size | Compressed |
| SD 1.5 | 49.49 M | – | – | 4×128×128 | 128 KB | – |
| SD 3.5 | 49.55 M | 3.0 MB | 1.4 MB | 16×128×128 | 512 KB | 277 KB |
| FLUX.1 | 49.55 M | 3.0 MB | 1.3 MB | 16×128×128 | 512 KB | 321 KB |

**(c)** Per-GPU latency and price [26, 44, 45].

| GPU | Denoise | Decode | Price |
|---|---|---|---|
| RTX 4090 | 6.23 s | 67.2 ms | ∼$2K |
| RTX 5090 | 4.49 s | 47.3 ms | ∼$3.8K |
| H100 PCIe | 4.09 s | 32.6 ms | ∼$25K |

smaller than the raw pixel tensor (512 KB vs. 3.0 MB). After lossless compression on both sides, the gap remains substantial: a pcodec-compressed latent occupies only ∼277 KB, still approximately 5× smaller than the corresponding PNG (∼1.4 MB).

**Decoding is much faster than generation.** Image generation is typically compute-bound because it requires iterative denoising; for SD 3.5, this stage takes 4–6 seconds per image, depending on the GPU. In contrast, decoding consists of a single deterministic forward pass and completes in only tens of milliseconds. Table 1c reports the per-image latency of both stages on three representative GPUs. *Decode latency remains well below the sub-second threshold typically associated with interactive image retrieval.*

**Consumer GPUs are cost-effective for decoding.** The decoding latency gap between datacenter and consumer GPUs is modest. Although RTX 5090 and RTX 4090 are 85% and 92% cheaper than H100, decoding a latent on RTX 5090 increases latency by less than 15 ms compared to H100, whereas RTX 4090 increases latency by another 20 ms as shown in Table 1c. As a comparison, fetching data from AWS S3 takes 100–200 ms [8]. This suggests that on-demand reconstruction can run efficiently on commodity GPUs and substantially reduces the cost per decode.

**Decoders are lightweight.** Table 1b compares decoders across three widely used model families. All three are compact neural networks with roughly 49.5 M parameters. At bfloat16 precision, each decoder occupies less than 100 MB of GPU memory, making it *practical to co-locate multiple decoders on a single GPU or to load a decoder on demand with minimal startup overhead.*

In summary, as the volume of generated images continues to grow rapidly, efficient and sustainable storage systems should exploit the distinctive properties of diffusion-based generation: storing less popular or older images as compact latent and reconstructing them only when needed.

# 4 LatentStore Design

This section presents the architecture of LatentStore, a distributed serving system that stores AI-generated images as compressed latent tensors and uses a dual-format cache to reduce serving latency and decoding cost. The design of LatentStore must address three coupled challenges.

**Challenge #1 (C1).** Although on-demand decoding adds only less than 50 ms to the critical path, it still incurs nontrivial compute overhead, so caching is necessary to avoid repeated decodes of popular content. Yet neither single-format extreme is sufficient. An all-image cache avoids decoding on hits but can cache only a limited number of objects, whereas an all-latent cache[2] maximizes the number of cached items, but forces a GPU decode on every hit. Because the workload combines a highly skewed popularity distribution (O1) with widely varying re-access intervals (O4), LatentStore must cache both formats at once.

**Challenge #2 (C2).** The optimal fraction of cache capacity devoted to decoded images is workload-dependent: rapid post-birth popularity decay (O2). The continual arrival of new images implies that the value of storing an image vs. a latent depends on the popularity and how long they are cached.

**Challenge #3 (C3).** Routing must simultaneously preserve cache locality and balance GPU load. Least-loaded routing spreads requests for the same image across nodes, leading to redundant caching of hot content and reducing effective aggregate capacity, while imbalanced load for latent wastes GPU cycles. LatentStore therefore requires a unified design that jointly manages the cache and request routing.

LatentStore addresses C1 with a *dual-format cache* (§4.3), C2 with *online marginal-hit tuning* of the image-to-latent ratio (§4.3), and C3 with *consistent-hashing routing* that uses a cache-pinned spillover (§4.4).

## 4.1 Design Overview

**Latent representation as a storage object.** The denoising process maps a text prompt to a compact latent tensor, and a subsequent lossless compression step shrinks it further, so a stored compressed latent is substantially smaller than the corresponding decoded image.

**System components.** Fig. 5 shows the architecture of LatentStore. A *frontend router* fronts a fleet of homogeneous GPU *nodes*, backed by a cloud object store (e.g., Amazon S3) that durably persists the compressed latents. The router maintains two pieces of lightweight runtime state: a map of in-flight image identifiers, so that a burst of identical requests issues only one decode, and a *routing table* that uses con-

[2]Caching latent reduces the latency of fetching from an object store (100-200 ms [8]).

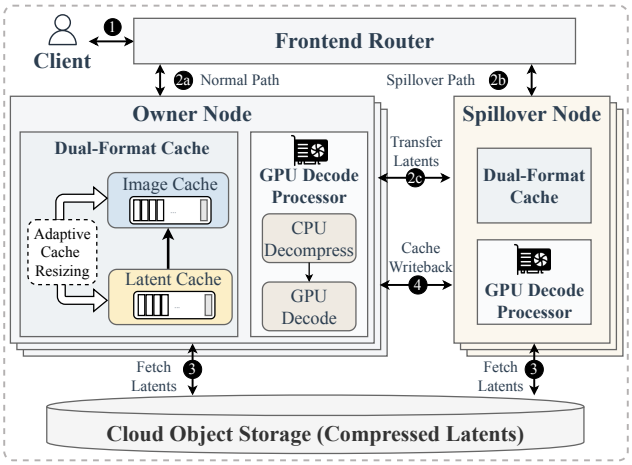



**Figure 5:** LatentStore architecture and request flow.

sistent hashing to map each identifier to its *owner node* and tracks per-GPU queue depths for load-aware dispatch. Every GPU node is functionally identical: it hosts a *dual-format cache* that holds each object either as a decoded image or as a compressed latent (§4.3), an *adaptive resizer* that balances the two tiers online (§4.3), and one *decode pipeline* per GPU streaming CPU decompression and GPU decode (§5).

**Request flow.** A client request (❶ in Fig. 5) first reaches the router, which checks whether the image is being decoded and applies consistent hashing to map the identifier to its owner node. Two paths follow:

- **Normal path** (❷a): The owner node is not overloaded, so the router dispatches directly. The node probes its cache: an image-tier hit returns immediately; a latent-tier hit triggers local GPU decode; a full miss fetches the compressed latent from cloud storage (❸) before decoding.
- **Spillover path** (❷b–❹): the owner node's GPU queue exceeds a threshold θ, so the router redirects the request to a less-loaded *spillover node*. If the owner already holds the latent, it is shipped to the spillover node (❷c, "share latents"); otherwise the spillover node fetches it from cloud storage (❸). After GPU decoding, the result is written back to the owner node's cache (❹), so the entry stays where consistent hashing placed it.

The cloud object store thus serves as the single source of truth for every object and is touched only on full cache misses; latents on average are about 20% the size of decoded images, dramatically reducing both storage footprint and network transfer cost compared with persisting full-resolution pixels (§6.2).

## 4.2 Dual-Format Cache

Caching decoded images reduces serving latency, but limited cache capacity restricts how many images can be stored. Caching compact latents allows many more items to fit in the cache (more coverage), though it requires GPU decoding. One option is to cache both images and latent items in a mixed-format LRU order. However, a given capacity cut-off contains an unpredictable mix of the two formats whose composition shifts with every access, making it infeasible to control the memory partition precisely.

LatentStore therefore maintains a *dual-format cache* on each node: two independent cache tiers, one for decoded images and one for compressed latents. The two caches share a fixed total capacity $C$. An adaptive cache resizer assigns an $\alpha$ fraction of $C$ to the *image cache* and the remaining $1-\alpha$ to the *latent cache*. In this way, each cache's capacity is fully determined by $\alpha$, and that each cache's miss ratio curve reflects a single, homogeneous object format, which is the prerequisite for the marginal tuning in §4.3. Fig. 6 illustrates the design. Table 2 summarizes the notation used throughout the following subsections.

**Lookup flow** The two caches have two operational characteristics. First, lookup is cascading: image cache first, then latent cache, and finally a cloud fetch on a full miss. Second, every object lives in exactly one cache at a time. A new object enters the latent cache on its first cloud fetch with a hit counter at zero; each subsequent latent-cache access increments the counter; once the counter crosses a promotion threshold $h$, the object is decoded, inserted into the image cache, and atomically removed from the latent cache.

**Per-access latency costs.** Every lookup resolves to one of three outcomes with sharply different costs. An image-cache hit returns the decoded image immediately and adds no latency. A latent-cache hit incurs $T_{decode}$, the GPU decode latency needed to reconstruct pixels from the compressed latent. A full miss incurs a cloud object-store fetch of the latent, with a cost at $T_{fetch}$, plus the same decode latency $T_{decode}$(Fig. 5).

## 4.3 Online Marginal-Hit Tuning

To address Challenge 2, LatentStore needs a mechanism to tune the image-to-latent ratio $\alpha$ online. A natural starting point is the *miss-ratio curve* (MRC) [16, 37, 56, 59]: given a cache of capacity $C$, the MRC maps $C$ to the miss ratio under a particular replacement policy. If the MRC were available for both caches, one could sweep over $\alpha$ and pick the split that minimizes overall latency. However, constructing full MRCs is impractical online. Moreover, the dual-format cache creates a *dependency* between the two caches: the latent cache sees only requests that missed the image cache, so its MRC changes whenever $\alpha$ changes. Globally minimizing latency would require the entire family of latent-cache MRCs indexed by image-tier size ($O(C^2)$), which is prohibitively expensive to maintain online via shadow caches or trace replay.

LatentStore sidesteps global MRC construction entirely. Instead of asking "what is the miss ratio at every possible capacity?", it asks a strictly local question at each tuning window: *which tier benefits more from one additional unit of capacity at the current operating point?* We call this the *marginal-hit* approach.

**Cost model and gradient-based tuning.** Let $\mathrm{MR}_{\mathrm{img}}(\alpha)$ denote the image-cache miss ratio at this allocation, measured

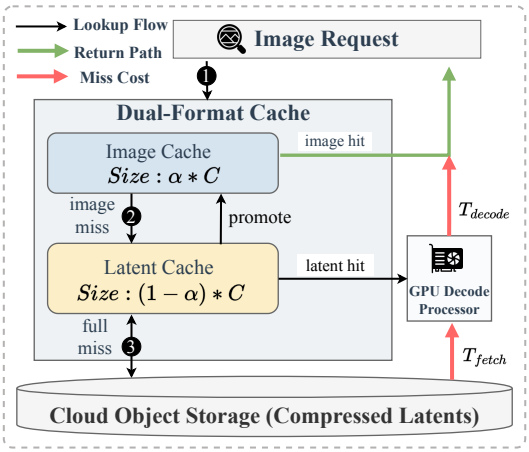



**Figure 6:** Dual-format cache design.

over the full request stream: a request counts as an *image miss* if the requested object is not found in the image cache, regardless of whether it is later found in the latent cache. Let $\mathrm{MR}_{\mathrm{lat}}(\alpha)$ denote the latent-cache miss ratio at this allocation, measured over the *image-miss stream*, i.e., only those requests that already missed the image cache. A latent miss is therefore a *full miss*: the object is absent from both caches and must be fetched from cloud storage. With cascading lookup (image cache → latent cache → cloud), the expected per-request cost is:

$$E[T](\alpha) = \underbrace{\big(1-\mathrm{MR}_{\mathrm{img}}(\alpha)\big)\cdot 0}_{\text{image-cache hit}} + \mathrm{MR}_{\mathrm{img}}(\alpha)\cdot\Big[\underbrace{\big(1-\mathrm{MR}_{\mathrm{lat}}(\alpha)\big)\cdot T_{decode}}_{\text{latent-cache hit}} + \underbrace{\mathrm{MR}_{\mathrm{lat}}(\alpha)\cdot\big(T_{decode}+T_{fetch}\big)}_{\text{full miss}}\Big] \tag{1}$$

Differentiating Eq. 1 with respect to $\alpha$ at the current operating point gives a scalar gradient $D$ whose sign directly prescribes the update direction:

$$D = -\delta_{img}\cdot\big[T_{decode} + T_{fetch}\,\mathrm{MR}_{\mathrm{lat}}(\alpha)\big] + T_{fetch}\,\mathrm{MR}_{\mathrm{img}}(\alpha)\cdot\delta_{lat}. \tag{2}$$

where $\delta_{img}$ measures how many additional image misses would appear if the image tier shrank by a small amount, and $\delta_{lat}$ measures how many additional full misses would appear if the latent tier were slightly smaller.

If $D<0$, the image tier has higher marginal value, so $\alpha$ increases by a fixed step $\Delta$; if $D>0$, the latent tier has higher marginal value, so $\alpha$ decreases by $\Delta$. No global sweep over $\alpha$ is needed: the sign of $D$ at the current operating point suffices for a single improving step.

**Online marginal measurement and cost update.** $\mathrm{MR}_I$ and $\mathrm{MR}_L$ are trivially counted along the lookup path. The marginal rates $\delta_{img}$ and $\delta_{lat}$ require slightly more structure. LatentStore splits each tier into a *main* segment of fraction $1-\tau$ and a thin *tail* segment of fraction $\tau$. Items evicted from the main enter the tail; items evicted from the tail leave the cache. A *tail hit* (a request served from the tail rather than the main segment) identifies a request that would have been a miss had that tier been $\tau$ smaller, so the tail-hit rate directly measures the marginal value of the last $\tau$ fraction of capacity. Concretely, all four statistics are accumulated over a window of $W$ requests, with hit, miss, and total-request counts all measured under the current partition at $\alpha$.

**Table 2:** Variables used in the dual-format cache and online marginal-hit tuning.

| Symbol | Definition |
|---|---|
| *Dual-format cache (§4.3):* | |
| $C$ | Per-node total cache capacity (bytes). |
| $\alpha$ | Fraction of $C$ given to the image cache; $1-\alpha$ to the latent cache. |
| $T_{decode}$ | GPU decode latency. |
| $T_{fetch}$ | Cloud fetch latency. |
| $h$ | Latent-to-image promotion threshold (hit counter). |
| *Online marginal-hit tuning (§4.3):* | |
| $E[T](\alpha)$ | Expected per-request latency at allocation $\alpha$. |
| $\mathrm{MR}_{\mathrm{img}}(\alpha)$ | Image-cache miss ratio over all requests. |
| $\mathrm{MR}_{\mathrm{lat}}(\alpha)$ | Latent-cache miss ratio, conditional on image-cache miss. |
| $\delta_{img}$ | Marginal image-miss rate (image-cache tail-hit fraction). |
| $\delta_{lat}$ | Marginal full-miss rate (latent-cache tail-hit fraction). |
| $D$ | Scalar gradient $\partial E[T]/\partial\alpha$; sign sets update direction. |
| $W$ | Tuning window size (number of requests). |
| $\Delta$ | Per-window step size applied to $\alpha$. |
| $\tau$ | Fraction of each cache tier reserved as the tail segment. |

$$\mathrm{MR}_{img}(\alpha) = \frac{\text{image misses}}{\text{total requests}}, \qquad \delta_{img}(\alpha) = \frac{\text{image tail hits}}{\text{total requests}},$$
$$\mathrm{MR}_{lat}(\alpha) = \frac{\text{full misses}}{\text{image misses}}, \qquad \delta_{lat}(\alpha) = \frac{\text{latent tail hits}}{\text{image misses}}.$$

After each window the gradient $D$ is re-evaluated, $\alpha$ is updated, tier capacities are recomputed, overflows are evicted, and all counters reset. This procedure adds *near-zero overhead*: the tail is carved from the existing cache budget without extra memory, and the four counters are maintained as side effects of the normal lookup path. No global MRC is ever reconstructed; the allocator moves $\alpha$ incrementally using only the gradient observed at the current operating point.

$T_{decode}$ and $T_{fetch}$ are tracked as exponentially weighted moving averages of observed GPU-decode and cloud-storage-fetch latencies. This closes a negative feedback loop that absorbs both workload shifts and transient infrastructure events with a single mechanism (Fig. 6). When the GPU is overloaded, $T_{decode}$ rises and the conditional cost of an image miss grows; the gradient pushes $\alpha$ upward to enlarge the image cache and offload GPU work. Conversely, when storage latency spikes, $T_{fetch}$ rises and the gradient shifts capacity toward the latent cache to broaden coverage and avoid full misses. Popularity drift, GPU throttling, and storage backpressure thus all feed into the same tuning loop without separate mechanisms.

## 4.4 Routing with Spillover

To address C3, LatentStore chains three router-side stages into a single mechanism that decouples *where the decode runs* from *where the cache entry lives*: *request coalescing* absorbs duplicate bursts, *consistent-hashing dispatch* pins

each item to a stable owner node, and *spillover with cache pinning* reconciles locality with GPU load by letting work move between nodes while every cache entry stays anchored at its hash-determined home.

**Request Coalescing.** The router maintains a map from image identifiers to in-flight decode requests. A new request for an in-flight identifier waits on the existing decode request rather than issuing a duplicate decode. For bursty workloads where the same image is requested hundreds of times per second, request coalescing substantially reduces the effective GPU decode load.

**Consistent-hashing routing.** The router uses consistent hashing to map each identifier to its owner node and, within that node, dispatches to the least-loaded GPU based on queue depths reported back to the router. Repeated accesses to the same item therefore land on the same owner node and reuse the same cached entry, eliminating cross-node duplication and maximizing aggregate cache effectiveness.

**Spillover with cache pinning.** When the least-loaded GPU on the owner node exceeds a queue-depth threshold $\theta$, the router activates the spillover path (2b–4 in Fig. 5). The central challenge is that the GPU work must move to a less-loaded node while the cache entry must remain on the hash-determined Owner Node. LatentStore resolves this by forwarding the decode task (along with cached latent if any) to the globally least-loaded spillover node and, once the decode completes, writing the result back to the owner node's cache.

This design guarantees two properties. First, *cache coherence*: every item's authoritative cache entry lives on exactly one node, regardless of which GPU did the decode. Second, *load spillover*: under extreme load LatentStore borrows GPU compute capacity from less-loaded nodes while preserving cache locality, rather than dropping requests or fragmenting cache state across nodes.

# 5 Implementation

We implemented LatentStore in Python atop Ray [39] for distributed actor management and TensorRT [43] for GPU-accelerated decoding.

**Distributed actor placement.** LatentStore maps its logical architecture onto Ray actors with node-affinity scheduling. Each physical GPU node hosts one *cache actor* that manages the node's dual-format cache, and one *decoder actor* per GPU. Placement constraints co-locate all actors for a node on the same machine, so cache lookups and latent transfers are local memory operations. The frontend router runs as an asynchronous event loop on the router node, issuing non-blocking remote calls to per-node actors.

**Pipelined GPU decoding.** LatentStore decomposes each decode into a multi-stage pipeline spanning three specialized thread pools: an *I/O pool* for cloud storage reads, a *compute pool* for CPU-bound decompression, host-to-device transfer, and image encoding, and a *GPU pool* that serializes TensorRT inference behind an asynchronous lock to ensure mutual exclusion on the device. This pipeline keeps the GPU busy while other requests concurrently fetch from storage, decompress latents, or encode output images, thereby improving throughput under concurrent load.

**TensorRT and CUDA Graphs.** The decoder is ahead-of-time compiled to a TensorRT engine with FP16 precision and a fixed input shape. At build time, LatentStore exports the PyTorch model to ONNX [1], applies TensorRT's layer-fusion and kernel-auto-tuning passes, and caches the resulting engine plan to disk for fast subsequent loads. At runtime, the engine is wrapped in a CUDA Graph [42] that captures the entire decode kernel sequence into a single launchable unit, eliminating per-decode kernel dispatch overhead and reducing decode latency compared to eager execution.

**Latent compression.** Compressed latents stored in cloud object storage are encoded with pcodec [32], a lossless compressor designed for columnar numeric arrays. We choose pcodec over general-purpose byte-stream compressors (e.g., zstd [22], LZ4 [34]) because diffusion-model latents are floating-point tensors with high spatial smoothness and inter-channel correlation, which byte-oriented entropy coders cannot exploit effectively. The resulting compact representation reduces both storage cost and network transfer time on cache misses.

# 6 Evaluation

Our evaluation answers four questions:

- **Q1:** What *data reduction ratio* does LatentStore achieve relative to storing images? (§6.2)
- **Q2:** Does LatentStore sacrifice read latency? (§6.3)
- **Q3:** What *long-term cost savings* does LatentStore provide over a multi-year horizon compared to image storage, and how does the picture shift across GPU price points? (§6.4)
- **Q4:** How important is each design choice, including dual-format caching, online tuning, and spillover dispatch, and how sensitive is LatentStore to parameter settings? (§6.5)
- **Q5:** How does LatentStore compare with lossy image compression? (§6.6)

## 6.1 Experimental Setup

**Testbed.** We deploy LatentStore on a cluster of three homogeneous GPU nodes, each equipped with one NVIDIA H100 80 GB PCIe GPU, an AMD EPYC 9554 CPU (28 vCPUs), 177 GB host memory, and 25 Gbps network connectivity. We use AWS S3 (us-east-1) as the backend storage and an EC2 c6in.4xlarge instance in the same region as the client. Each GPU node is allocated a 2 GB dual-format cache, corresponding to 1% of the total unique-object footprint. We chose H100 GPUs for their wide availability on cloud platforms; consumer GPUs (e.g., RTX 5090) were unavailable for rent on major cloud providers at the time of our experiments.

**Dataset.** We generate 150 K images at $1024 \times 1024$ resolution using Stable Diffusion 3.5 [19] and store both the pixel PNGs and pcodec-compressed latents in S3. The full dataset occupies 210.6 GB as PNG and 41.5 GB as compressed latents. To

derive a replay workload from the CompanyX trace (§3.1.1), we randomly sample 150 K unique object IDs (∼615× object-level downsample), preserving all accesses to the sampled objects, and map each to one of our generated SD 3.5 images. The resulting *downsampled trace* contains 57.2 M requests and is used for sensitivity analysis in §6.5.

For the end-to-end latency evaluation (§6.3), we select a contiguous 48-hour window from the downsampled trace and replay it at 10× wall-clock speed, preserving the original request ordering. This *evaluation window* contains 66,581 requests targeting 12,328 unique objects whose aggregate PNG footprint is 21.4 GB (3.6 GB as compressed latents). Before evaluation, we pre-warm each node's cache with requests prior to the evaluation requests.

**Baselines and ablations.** We evaluate six configurations spanning on-demand generation, traditional image store, and LatentStore variants.

*Upper-bound reference:*

- **On-Demand Generation**: every previously unseen request triggers the full SD 3.5 diffusion pipeline with 28 denoising steps on a GPU. A local image cache retains previously generated results to return repeat reads without re-generation. This configuration establishes the cost of on-demand generation and is evaluated on a 1,000-request subset using the same 3-node GPU cluster.

*Store-and-read configurations:*

1. **Decode-All**: every request fetches a compressed latent from S3 and performs GPU decode. No local caching is used, establishing the raw read latency floor.
2. **ImgStore**: a distributed image server backed by S3-stored PNG images with a 2 GB per-node LRU cache. Cache hits return pre-decoded bytes with no GPU involvement; misses fetch the full PNG from S3.
3. **LS-ImgCache** ($\alpha$=1.0): LatentStore with an image-only cache; the entire cache budget stores decoded images. Misses fetch and decode compressed latents from S3.
4. **LS-LatentCache** ($\alpha$=0.0): LatentStore with a latent-only cache; the cache stores only compressed latents. Every cache hit still requires GPU decode.
5. **LS-Adaptive**: LatentStore with the dual-format cache and marginal-hit online tuning, adaptively splitting each node's budget between an image tier and a latent tier.

**Metrics.** For storage, we report the *data reduction ratio* (DRR), defined as the fraction of PNG bytes eliminated: $\text{DRR} = (S_{\text{PNG}} - S_{\text{comp}})/S_{\text{PNG}}$, where $S_{\text{comp}}$ is the compressed-latent footprint under LatentStore. For read access, we report mean, P50, P95, and P99 *read latency*, defined as the wall-clock time from request arrival at the frontend router to response completion. This metric covers the entire in-cluster processing pipeline, including cache lookup, S3 fetch (on miss), GPU decode, GPU queuing, and data transfer back to the router, but excludes the variable client-to-cluster network hop. For cache-level analysis, we also report image-tier hit rate, latent-tier hit rate, and full-miss rate.

**Table 3:** Storage footprint comparison across models and resolutions. ImgStore is the baseline image storage; LatentStore: LS; LS stores full pcodec-compressed format.

| Model | Res. | #Imgs | ImgStore | LS | DRR (%) |
|---|---|---|---|---|---|
| SD 3.5 | 1024 × 1024 | 150 K | 210.6 GB | 41.5 GB | 80.3 |
| SD 3.5 | 512 × 512 | 150 K | 57.1 GB | 10.9 GB | 80.8 |
| FLUX.1 | 1024 × 1024 | 100 K | 130.4 GB | 32.1 GB | 75.4 |
| FLUX.1 | 512 × 512 | 100 K | 35.9 GB | 8.0 GB | 77.6 |
| **Total** | | **500 K** | **434.1 GB** | **92.6 GB** | **78.7** |

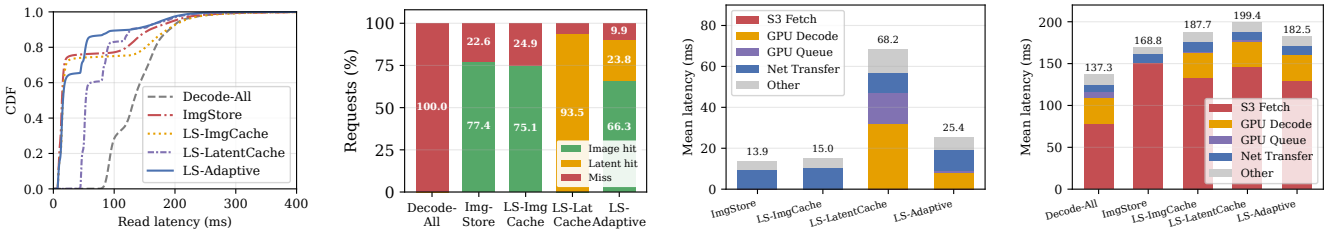
## 6.2 Storage Reduction

Table 3 compares ImgStore to LatentStore without compression and to pcodec-compressed latents under LatentStore, for Stable Diffusion (SD) 3.5 and FLUX.1 at 512×512 and 1024×1024. We generate 150 K images with SD 3.5 and 100 K images with FLUX.1 at each resolution, totalling 500 K images. Relative to ImgStore, LatentStore attains an aggregate *DRR* of **78.7%**: ImgStore occupies 434.1 GB while LatentStore occupies only 92.6 GB for the same corpus. Per row, *DRR* spans 75.4%–80.9%. Note that the ImgStore baseline itself is already losslessly compressed; compared to uncompressed RGB pixels at three bytes per pixel, the *DRR* would be even larger: a single 1024×1024 image is 3 MB as RGB, whereas LatentStore stores it in ∼0.29 MB. Two mechanisms contribute to the *DRR* relative to ImgStore. First, the encoder maps high-resolution pixel grids into compact latent tensors, reducing spatial dimensions while introducing a modest number of latent channels. This encoding alone removes 64–68% of the ImgStore footprint, shrinking the corpus from 434.1 GB down to 152.6 GB across both models. Second, pcodec lossless compression yields an additional data reduction ratio of 34–43% relative to the uncompressed latent tensors, exploiting the numerical structure of those tensors to reach 92.6 GB.

Notably, SD 3.5 latents are more compressible than FLUX.1 latents, with an average pcodec DRR of 0.53 and 0.34, respectively, likely due to differences in their respective VAE architectures and latent-space distributions. The DRR remains stable across resolutions for both models, indicating that LatentStore's storage benefits scale predictably with image size. At datacenter scale, the same *DRR* implies large absolute byte savings: storing 100 billion 1024×1024 SD 3.5 images would require ∼140 PB as ImgStore but only ∼30 PB under LatentStore.

## 6.3 End-to-End Serving Performance

We evaluate LatentStore's read latency under the full 48-hour trace replay on the prototype cluster. All store-and-read configurations described in §6.1 use the same trace and 720-hour cache-warmup procedure; the only differences are the cache allocation policy and, for ImgStore, the stored data format (S3 images vs. S3 latents).




**(a)** Read latency CDF. **(b)** Cache hit distribution. **(c)** Hit latency breakdown. **(d)** Miss latency breakdown.

**Figure 7:** End-to-end read performance. (a) CDF of read latency for five store-and-read configurations. (b) Stacked cache hit distribution: image hit, latent hit, and full miss fractions. (c) Mean latency breakdown for cache-hit requests (image hit + latent hit); numbers above bars show total mean (ms). (d) Mean latency breakdown for cache-miss requests.

**Table 4:** Read latency (ms). Store-and-read configurations use 3 nodes × 2 GB per-node cache (6 GB total, 66,581 requests). Generation latency is measured separately over 1,000 requests.

| Configuration | Mean | P50 | P95 | P99 |
|---|---|---|---|---|
| Generation | 4,309 | 4,130 | 13,647 | 17,518 |
| Decode-All | 137.3 | 132.6 | 209.1 | 275.9 |
| ImgStore | 49.4 | **14.1** | 198.7 | 276.9 |
| LS-ImgCache | 58.0 | 15.6 | 213.9 | 285.3 |
| LS-LatentCache | 77.0 | 53.8 | 180.9 | 234.3 |
| **LS-Adaptive** | **41.2** | 15.8 | **178.5** | **227.3** |

#### 6.3.1 Generation vs. Store-and-Read

On-demand SD 3.5 generation takes 3,905 ms per image on a single H100, yielding a mean of 4,309 ms in trace replay (Table 4). The mean latency of Decode-All is 137 ms, a 31× reduction; LatentStore further reduces this to 41.2 ms, a 105× reduction over generation.

#### 6.3.2 Overall Read Latency

Fig. 7a and Table 4 compare the five store-and-read configurations. ImgStore achieves a P50 of 14.1 ms, comparable to LatentStore's 15.8 ms, as both share the same 3-node topology and image cache hits avoid GPU decode. However, cache misses (22.6%) in ImgStore require a full S3 PNG round trip, driving the P99 to 277 ms and the mean to 49.4 ms. LS-ImgCache has a comparable miss rate (24.9%), but each miss additionally incurs GPU decode after the S3 latent fetch, yielding a higher mean (58 ms). LS-LatentCache minimizes misses (6.5%) by fitting ∼5× more objects into the same budget, but every hit still requires GPU decode (∼31 ms). This shifts the entire LS-LatentCache CDF rightward by an almost constant decode-and-encode latency relative to LatentStore's image-hit fast path; under sustained load, GPU queue contention pushes its mean to 77 ms, the highest among cached configurations.

LatentStore combines both tiers: popular objects are read as zero-decode image retrieval (∼15 ms), with the first inflection point corresponding to 66.3% of pixel image hits (Fig. 7b). The second segment of the CDF curve corresponds to latent-tier hits, incurring decoding cost and leaving only 9.9% of requests to traverse the full S3 + decode pipeline. The result is a mean of 41.2 ms, 17% lower than ImgStore (49.4 ms), and a P99 of 227 ms, 18% below ImgStore's 277 ms. The advantage stems from two factors: (1) fewer misses reach S3 at all, and (2) misses fetch compact latents (∼0.28 MB) rather than full PNGs (∼1.4 MB), yielding shorter and less variable S3 transfer latency.

#### 6.3.3 Latency Breakdown and Cache Hit Distribution

We separate hit and miss requests and shows where latency is spent in each case. Net Transfer, the time to stream the final PNG from the GPU node back to the frontend router, is a near-constant ∼10 ms across all configurations and hit types. For cache hits (Fig. 7c), image hits return in ∼15 ms with negligible compute, dominated by network transfer alone; latent hits add ∼31 ms of GPU decode. For misses (Fig. 7d), S3 fetch dominates at 79–146 ms, dwarfing the 31 ms GPU decode. Notably, Decode-All exhibits the lowest per-miss S3 latency of 79 ms mean despite issuing the most S3 requests. Because every request in Decode-All reaches S3, including frequently accessed objects, S3's internal caching layers keep popular objects warm, reducing average transfer time. In cached configurations, local caches absorb the popular objects, so only cold, long-tail objects reach S3, and these cold fetches see higher and more variable latency of 130–146 ms. Despite the higher per-miss cost, LatentStore's miss rate of 9.9% is less than half of ImgStore's 22.6%, yielding the lowest overall mean.GPU queue contention penalizes LS-LatentCache: when 93.5% of requests need decode, the mean queue wait reaches 15.2 ms (Fig. 7c), compared to 1.0 ms for LatentStore where only 33.7% touch the GPU.

### 6.4 Long-Term Cost Projection

This section quantifies LatentStore's long-term economics at scale. We replay the full 35-month CompanyX trace through a cost model and then extrapolate the steady state to 2050, comparing four setups across two GPU price points.

**Cost model.** We count two cost components that differ across strategies: *persistent storage* and *on-demand GPU decoding*. Common operational costs are omitted. Let $N(t)$ be the cumulative number of images at time $t$, $\bar{S}_{\text{px}}$ and $\bar{S}_{\text{lat}}$ the average per-PNG image size and compressed latent respectively, $f$ the pixel-cache fraction of the working set, $M(t)$ the decode count, and the per-decode cost $P_{\text{dec}} = t_{\text{dec}} \cdot P_{\text{GPU}}$. Then

**Table 5:** Cost model parameters.

| Parameter | Value | Notes |
|---|---|---|
| $\bar{S}_{\text{px}}$ | 1.5 MB | Average PNG (1024 × 1024) |
| $\bar{S}_{\text{lat}}$ | 0.29 MB | Compressed latent (SD 3.5) |
| $P_{\text{S3}}$ | \$0.023/GB-mo | AWS S3 Standard [5] |
| $P_{\text{Glacier}}$ | \$0.004/GB-mo | S3 Glacier IR storage (objects ≥5 yr) [6] |
| $P_{\text{GIR-ret}}$ | \$0.01/GB + \$0.0001/req | S3 Glacier IR retrieval |
| $P_{\text{H100}}$ | \$2.50/GPU-hr | Datacenter GPU rental [26] |
| $P_{5090}$ | \$0.69/GPU-hr | Consumer-class GPU rental [23] |
| $t_{\text{dec}}$ | 40 ms | SD 3.5 decode |
| $f$ | 1% | Dual-format cache fraction of working set |
| $m_{\text{gpu}}$ | 63.2% | LatentStore decode-trigger rate |
| $\lambda$ | 10.2/yr | Mean views per image |

$$C_{\text{ImgStore}}(t) = N(t) \cdot \bar{S}_{\text{px}} \cdot P_{\text{S3}} \quad (3)$$

$$C_{\text{LatentStore}}(t) = \underbrace{N(t) \cdot (\bar{S}_{\text{lat}} + f \cdot \bar{S}_{\text{px}}) \cdot P_{\text{S3}}}_{\text{latent + pixel-cache storage}} + \underbrace{M(t) \cdot P_{\text{dec}}}_{\text{GPU decode}} \quad (4)$$

Table 5 lists all parameters. We compare four setups: ImgStore on S3 Standard; ImgStore with a 5-year Glacier IR archive tier, where objects older than 5 years migrate to S3 Glacier IR at \$0.004/GB-mo and retrieval cost is estimated via the stratified age-decay model fitted on the trace; and LatentStore with a 1% dual-format cache, where the empirically measured $m_{\text{gpu}}$=63.2% of requests trigger a decode, evaluated at two GPU rental rates: H100 at \$2.50/GPU-hr and RTX 5090 at \$0.69/GPU-hr.

For the projection, we extrapolate the steady state observed in the trace tail: the platform adds ∼3.76 M new images per month, equivalent to a 12.7% CAGR of the cumulative working set over the projection horizon. Monthly decode demand is $m_{\text{gpu}} \cdot \lambda \cdot N(t)/12$, holding $m_{\text{gpu}}$ and $\lambda$ constant. Fig. 8 compares the resulting cumulative cost at four time horizons, normalized so that ImgStore at trace end in March 2026 equals 1.

**Cost during the trace period.** Over the 35-month trace window, LS-Adaptive on 5090 accumulates only 0.40× the ImgStore total, a **60% saving**. LS-Adaptive on H100 reaches 0.91× for a 9% saving, as the higher GPU rental rate narrows the advantage on short horizons where the cumulative storage gap is still modest.

**Long-term projection.** By 2050, storage cost has accumulated over a linearly growing working set, and LatentStore's structural storage advantage becomes decisive. Under constant prices (Fig. 8-top), ImgStore reaches 164× its 2026 value, while LS-Adaptive on 5090 reaches 49×, a **70% saving**. Even on H100, LS-Adaptive reaches 88× for a 46% saving. A natural strengthening of the ImgStore baseline is to tier old data to Glacier Instant Retrieval (IR) [7]. With a 5-year archive cutoff and retrieval cost modeled via the stratified age-decay fit from O2, ImgStore + Glacier IR reaches 79× by 2050. LS-Adaptive on 5090 still saves 39% relative to this tiered baseline at 49× vs. 79×.

Under a more optimistic price-decline scenario (GPU −20%/yr, storage −10%/yr from 2026; Fig. 8-bottom), the gap widens further: ImgStore reaches 40×, while LS-Adaptive on 5090 reaches only 9.7×, a **75% saving**. Even compared to ImgStore + Glacier IR at 27×, LS-Adaptive on 5090 saves 64%, because cheaper GPUs amplify the decode-based architecture's advantage while storage-price reductions benefit both strategies proportionally.

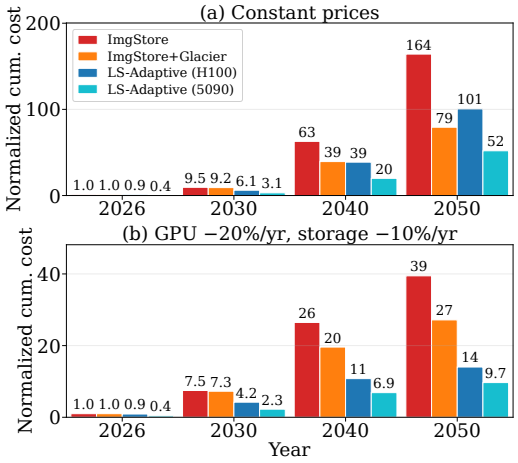



**Figure 8:** Cumulative cost at four time horizons (2026, 2030, 2040, 2050), normalized so ImgStore at the trace end (March 2026) equals 1. (Top) constant prices. (Bottom) with annual price decay (GPU −20%/yr, storage −10%/yr from 2026 [17, 18, 38, 49]).

## 6.5 Ablation and Sensitivity Analysis

We use both prototype-cluster experiments and trace-driven simulation to validate each of LatentStore's three core design choices and to characterize parameter sensitivity:

1. **Dual-format caching** vs. single-format baselines (§6.5.1);
2. **Online ratio tuning** vs. static allocation (§6.5.2);
3. **Spillover dispatch** vs. hash-only routing (§6.5.3);
4. **Parameter sensitivity** of the tuning algorithm (§6.5.4).

For the simulation experiments, the simulator faithfully implements the dual-format cache with independent LRU tiers, the main/tail split, and the marginal-hit gradient update rule. The tuning algorithm uses six parameters: $T_{decode}$ and $T_{fetch}$ are the per-request latency costs of a latent-tier hit (GPU decode) and a full miss (S3 fetch + decode), respectively; $\Delta$ is the step size by which $\alpha$ is adjusted at each window boundary; $W$ is the number of requests per gradient-estimation window; $\tau$ is the fraction of each LRU tier reserved as a *tail* segment for marginal hit-rate estimation; and $h$ is the promotion threshold, i.e., the number of latent hits an object must accumulate before being promoted to the image tier. Unless otherwise noted, simulations use $T_{decode}$=40 ms, $T_{fetch}$=140 ms, $\Delta$=0.005, $W$=1,000,000, $\tau$=0.10, and $h$=8.

### 6.5.1 Dual-Format Cache across Cache Sizes

Table 6 compares latency for LatentStore's dual-format cache against its two single-format variants, LS-ImgCache ($\alpha$=1) and LS-LatentCache ($\alpha$=0), across six cache sizes. At the smallest cache size (0.1%), LS-LatentCache is competitive because compressed latents are ∼5× smaller than PNGs, fitting more objects into the same budget. As the cache grows, repeated GPU decode on popular items becomes the bottleneck; LS-ImgCache surpasses LS-LatentCache starting at

**Table 6:** Latency (ms) across cache sizes (% of WSS) for LatentStore's dual-format cache vs. single-format baselines.

| | Cache size (% of WSS) | | | | | |
|---|---|---|---|---|---|---|
| **Configuration** | **0.1%** | **0.5%** | **1%** | **2%** | **5%** | **10%** |
| LS-ImgCache | 103.1 | 73.9 | 61.0 | 48.7 | 33.7 | 23.2 |
| LS-LatentCache | 97.5 | 74.9 | 66.2 | 58.0 | 50.1 | 47.0 |
| **LS-Adaptive** | **90.1** | **62.2** | **50.7** | **40.1** | **28.5** | **21.9** |

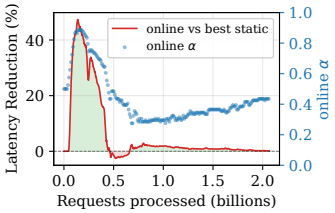



**Figure 9:** Per-window latency improvement and α trajectory.

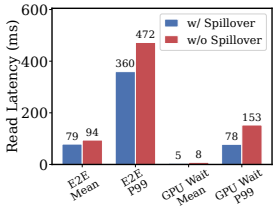



**Figure 10:** Latency with and without spillover dispatch.

0.5%. The best single-format policy therefore changes with cache size, confirming that no single format dominates. LatentStore's dual-format cache achieves the **lowest** latency at every capacity, outperforming the best single-format alternative by 5–18%.

### 6.5.2 Online Tuning vs. Static Allocation

A natural question is whether a *fixed* α, picked offline, could match the adaptive resizer. We sweep $\alpha \in \{0.3, 0.4, 0.5, 0.6, 0.7\}$ on the full 2.07 B-request trace; the lowest latency among all static choices is α=0.5 at 51.6 ms. LatentStore's adaptive resizer reaches **50.7 ms**, 1.7% lower than this oracle-picked best static, and matches or beats α=0.5 in **87%** of all 1 M-request windows, despite being given no prior knowledge of the trace.

Fig. 9 shows where the gain comes from. During the first 0.4 B requests, the 1% budget already covers most of the small catalog of up to 4 M images, so format choice dominates over capacity. LatentStore raises α to 0.7–0.85 to prioritize image hits that skip the 40 ms decode, while any fixed α≤0.5 wastes half its slots on latents that still pay it; the gain peaks at **30–45%**. Once the catalog grows past 60 M, misses dominate latency regardless of α and the gain settles into a steady-state **1–3%** lead, with α pulled back to 0.3–0.4. The single brief dip below zero near 0.5 B requests is the phase transition where the catalog first outgrows the cache and the controller takes a few windows to re-converge.

The practical message is twofold. First, even on this stationary trace where a single static α is competitive, the adaptive resizer is strictly better on average and never asks the operator to guess. Second, the cost of guessing wrong is real: α=0.3 and α=0.7 are both within 2.4% of the best static on this trace, but would drift away from the optimum on a workload whose hit-cost distribution is even modestly different, for example after a model migration that changes $T_{decode}$ or a content surge that shrinks effective cache coverage. LatentStore absorbs that risk automatically.

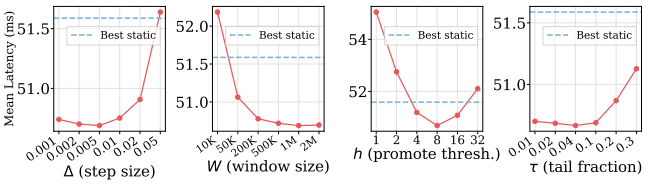



**Figure 11:** Sensitivity analysis. LatentStore is robust to step size, window size, and tail fraction; the promotion threshold $h$ has the largest impact.

### 6.5.3 Spillover Dispatch

To validate the effectiveness of the spillover path, we replay the same 48-hour trace on a 6-node GPU cluster at 1000× speed and an overflow threshold θ=4. The *without-spillover* baseline sets θ to infinity, so every request is dispatched to its hash-determined owner node regardless of queue depth. Fig. 10 compares latency with and without spillover. Spillover reduces mean E2E latency by **16.5%** (94.5 ms to 78.9 ms) and P99 by **23.9%** (472.5 ms to 359.5 ms).

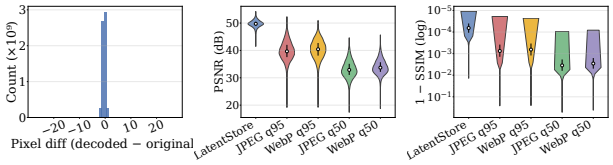
The gain is concentrated in GPU queue wait time, where P99 drops by 49% from 152.8 ms to 78.0 ms, confirming that spillover alleviates head-of-line blocking on hot nodes. Under consistent-hashing routing, natural load skew causes some nodes to accumulate deeper queues; spillover detects this via the threshold θ and redirects excess work to idle nodes while keeping cache entries pinned to their owner nodes, preserving locality without sacrificing tail latency.

### 6.5.4 Parameter Sensitivity

Fig. 11 examines the sensitivity of the online tuning to its four main parameters: step size Δ, window size $W$, promotion threshold $h$, and tail fraction τ. We sweep each parameter across a range while holding the others at their defaults (Δ=0.005, $W$=1M, τ=0.10, $h$=8), and report the latency on the full CompanyX trace at 1% cache size.

The results show that LatentStore is robust to parameter choices across three of the four parameters. **Step size** Δ (0.001–0.05): latency varies within 1.9% of the best; values from 0.001 to 0.02 are nearly identical (<0.5% spread), with only Δ=0.05 showing a noticeable penalty. **Window size** $W$ (10K–2M): large windows (≥200K) achieve near-optimal latency (within 0.2%); smaller windows ($W$=10K) incur a 3% penalty due to noisier gradient estimates. **Tail fraction** τ (0.01–0.3): the most insensitive parameter, with the entire range spanning only 0.9%. **Promotion threshold** $h$ has the largest effect (8.6% spread). Small values ($h$=1) promote aggressively, consuming image-cache capacity on rarely reused items; $h$=8 is optimal, and $h$≥4 brings latency within 1% of the minimum. Overly large values ($h$=32) delay promotion excessively, increasing latent-tier decode cost.

The insensitivity of Δ, $W$, and τ follows from the gradient's self-correcting nature: an overly large step overshoots the optimal α but is corrected in the next window. These results




**(a)** Pixel diff distribution. **(b)** PSNR distribution. **(c)** SSIM distribution.

**Figure 12:** Reconstruction fidelity over 10 K SD 3.5 images (1024×1024). (a) Signed per-channel pixel difference aggregated across 2,000 sampled images; 47% of pixel-channel values are unchanged. (b)-(c) White dot: median; thick bar: interquartile range. "q" in "q95" denotes quality factor; higher PSNR (dB) and SSIM closer to 1 indicate better fidelity.

indicate that practitioners can deploy LatentStore without extensive parameter tuning.

## 6.6 Comparing with Lossy Compression

**Floating-point decode determinism.** LatentStore preserves the original latent tensor bit-exactly via lossless pcodec compression. Since VAE decoding is a deterministic function of its input, the same latent on the same hardware and software stack always produces an identical pixel output. In practice, however, pixel-level differences can arise when the decoding environment changes because variations in GPU architecture or math library versions alter fused multiply-add ordering, introducing rounding noise at the level of individual floating-point operations. We performed decoding using 2,000 latents on H100 and L4 GPU, Fig. 12a shows that 47% of all pixel-channel values are bit-exact, and among those that differ, the vast majority shift by only ±1–3 levels out of 0–255, yielding a symmetric, zero-centered distribution that confirms the deviations are unbiased hardware rounding noise rather than information loss.

**Comparison with lossy codecs.** Because of rounding noise, we further compare LatentStore against standard lossy codecs at comparable file sizes using two widely adopted image fidelity metrics: PSNR (Peak Signal-to-Noise Ratio) [24], which measures per-pixel reconstruction error in decibels, and SSIM (Structural Similarity Index) [25], which captures perceived structural similarity on a 0–1 scale. LatentStore's compressed latent averages 290 KB per image (vs. 1,473 KB for the original PNG), comparable to JPEG q95 (333 KB) and WebP q95 (260 KB). At these sizes, LatentStore achieves a mean PSNR of 49.7 dB, roughly 10 dB higher than the best lossy codec (WebP q95 at 40.3 dB), corresponding to an order-of-magnitude reduction in mean squared error (Fig. 12b). SSIM tells the same story: LatentStore's mean SSIM is 0.9997, while WebP q95 reaches 0.9986 (Fig. 12c). Equally important, LatentStore's quality distribution is tightly concentrated (PSNR interquartile range $< 3$ dB), whereas lossy codecs exhibit a long lower tail where images with fine textures or sharp edges suffer disproportionate degradation. Lower-quality settings (JPEG/WebP q50) achieve smaller sizes (65–88 KB) but at the cost of 16–17 dB lower PSNR, making them unsuitable for archival storage of generated content.

# 7 Related Work

LatentStore lies at the intersection of large-scale image storage, generative-model optimization, computation–storage tradeoffs, and cache management. We discuss how it relates to and departs from prior work in each area.

**Large-scale image storage systems.** Conventional object storage systems [9, 10, 12, 40, 41, 64] have been extensively studied for serving billions of blobs. However, these systems treat images as opaque pixel blobs, whereas LatentStore stores compact latents and reconstructs pixels on demand.

**Diffusion model inference optimizations.** Extensive research accelerates image generation via efficient sampling [31, 53], optimized architectures [13, 27], and inference-time scaling [35]. Recent work further reduces latency through few-step distillation [33, 62], feature caching [36], and quantization [63]. However, these generation-stage optimizations—mapping noise to latents—are orthogonal to LatentStore. Faster generation does not alleviate the compounding storage burden of retaining billions of synthetic images. LatentStore operates strictly *after* generation, focusing instead on efficiently storing, caching, and reconstructing the resulting latent tensors for serving.

**Cache management and adaptive partitioning.** A line of work optimizes co-managing compressed and uncompressed data in the same memory cache [54,58,60]. Google's software-defined far memory [30] and Meta's TMO [57] demote cold pages to a compressed tier in datacenters and pair a hot uncompressed tier with a cold compressed tier sized by page-pressure signals, trading CPU decompression for a larger effective footprint. Unlike these systems, LatentStore does not tier the same data representation across compressed and uncompressed forms, but instead manages two distinct storage formats—decoded images and compressed latents.

# 8 Conclusion

This paper presents LatentStore, a novel latent-first storage system. LatentStore stores images as compressed latents and reconstructs pixels on demand using sparse GPU decoding, trading inexpensive compute for large persistent storage savings. To keep decode overhead low, LatentStore uses a small dual-format cache that balances an image LRU tier for fast hits against a latent LRU tier for more effective capacity. Evaluated with a large-scale production trace, LatentStore substantially reduces persistent storage cost while also lowering mean and tail read latency. More broadly, this work points to a shift in how generated content should be stored. As generative platforms continue to scale and GPU cost continues to fall, the compute-for-storage tradeoff offers a practical path toward more sustainable media storage systems.

## References


[1] ONNX: Open neural network exchange. https://onnx.ai, 2019.

[2] Midjourney. https://www.midjourney.com, 2026. Accessed: April 2026.

[3] Adobe. Introducing firefly foundry. https://business.adobe.com/blog/introducing-firefly-foundry, 2023. Accessed: 2026-04.

[4] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, Santa Clara, CA, July 2024. USENIX Association.

[5] Amazon Web Services. Amazon S3 pricing. Online, 2024.

[6] Amazon Web Services. Amazon S3 Glacier instant retrieval storage class, 2026. Accessed: 2026-04-23.

[7] Amazon Web Services. Amazon S3 Glacier Instant Retrieval storage class. https://aws.amazon.com/s3/storage-classes/glacier/instant-retrieval/, 2026. Accessed: 2026-04.

[8] Amazon Web Services. Best practices design patterns: Optimizing amazon s3 performance. https://docs.aws.amazon.com/AmazonS3/latest/userguide/optimizing-performance.html, 2026. Accessed: 2026-04-23.

[9] Shobana Balakrishnan, Richard Black, Austin Donnelly, Paul England, Adam Glass, Dave Harper, Sergey Legtchenko, Aaron Ogus, Eric Peterson, and Antony Rowstron. Pelican: A building block for exascale cold data storage. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 351–365, 2014.

[10] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[11] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[12] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157, 2011.

[13] Junsong Chen, Jincheng Yu, Chongjian Ge, Lewei Yao, Enze Xie, Yue Wu, Zhongdao Wang, James Kwok, Ping Luo, Huchuan Lu, and Zhenguo Li. PixArt-α: Fast training of diffusion transformer for photorealistic text-to-image synthesis. In *International Conference on Learning Representations (ICLR)*, 2024.

[14] Junyu Chen, Han Cai, Junsong Chen, Enze Xie, Shang Yang, Haotian Tang, Muyang Li, Yao Lu, and Song Han. Deep compression autoencoder for efficient high-resolution diffusion models. *arXiv preprint arXiv:2410.10733*, 2024.

[15] Mehdi Cherti, Romain Beaumont, Ross Wightman, Mitchell Wortsman, Gabriel Ilharco, Cade Gordon, Christoph Schuhmann, Ludwig Schmidt, and Jenia Jitsev. Reproducible scaling laws for contrastive language-image learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2818–2829, 2023.

[16] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, 2016.

[17] DiskPrices.com. Disk prices — current hard drive cost per gigabyte. https://diskprices.com/, 2025. Accessed 2026-04.

[18] Epoch AI. Data on machine learning hardware. https://epoch.ai/data/machine-learning-hardware, 2024. Dataset of >170 AI accelerators (GPUs, TPUs) with performance, price, and efficiency metrics. CC-BY 4.0. Accessed 2026-04.

[19] Patrick Esser, Sumith Kulal, Andreas Blattmann, Rahim Entezari, Jonas Müller, Harry Saini, Yam Levi, Dominik Lorenz, Axel Sauer, Frederic Boesel, et al. Scaling rectified flow transformers for high-resolution image synthesis. In *Forty-first international conference on machine learning*, 2024.

[20] Patrick Esser, Robin Rombach, and Bjorn Ommer. Taming transformers for high-resolution image synthesis. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12873–12883, 2021.

[21] Everypixel Journal. Ai image statistics. https://journal.everypixel.com/ai-image-statistics, 2023. Accessed: 2026-04.

[22] Facebook. Zstandard - Fast real-time compression algorithm. https://github.com/facebook/zstd. Accessed: 2026-04.

[23] GetDeploying. NVIDIA RTX 5090 gpu guide and pricing. https://getdeploying.com/gpus/nvidia-rtx-5090, 2026. Accessed: 2026-04.

[24] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Pearson, 4th edition, 2018.

[25] Alain Hore and Djemel Ziou. Image quality metrics: Psnr vs. ssim. In *2010 20th international conference on pattern recognition*, pages 2366–2369. IEEE, 2010.

[26] JarvisLabs. NVIDIA H100 price guide 2026: GPU costs, cloud pricing & buy vs rent. https://jarvislabs.ai/blog/h100-price, 2026. Accessed: 2026-04.

[27] Tero Karras, Miika Aittala, Timo Aila, and Samuli Laine. Elucidating the design space of diffusion-based generative models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[28] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[29] Black Forest Labs, Stephen Batifol, Andreas Blattmann, Frederic Boesel, Saksham Consul, Cyril Diagne, Tim Dockhorn, Jack English, Zion English, Patrick Esser, Sumith Kulal, Kyle Lacey, Yam Levi, Cheng Li, Dominik Lorenz, Jonas Müller, Dustin Podell, Robin Rombach, Harry Saini, Axel Sauer, and Luke Smith. Flux.1 kontext: Flow matching for in-context image generation and editing in latent space, 2025.

[30] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thorat, Adrian Yurtsever, Daniel Zolnowski, Kim Hazelwood, Martin Maas, Thomas Mccauley, and Rohit Sen. Software-defined far memory in warehouse-scale computers. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 317–330, 2019.

[31] Luping Liu, Yi Ren, Zhijie Lin, and Zhou Zhao. Pseudo numerical methods for diffusion models on manifolds. In *International Conference on Learning Representations (ICLR)*, 2022.

[32] Martin Loncaric, Niels Jeppesen, and Ben Zinberg. Pcodec: Better compression for numerical sequences, 2025.

[33] Simian Luo, Yiqin Tan, Longbo Huang, Jian Li, and Hang Zhao. Latent consistency models: Synthesizing high-resolution images with few-step inference. *arXiv preprint arXiv:2310.04378*, 2023.

[34] lz4. LZ4 - Extremely fast compression. https://github.com/lz4/lz4. Accessed: 2026-04.

[35] Nanye Ma, Shangyuan Tong, Haolin Jia, Hexiang Hu, Yu-Chuan Su, Mingda Zhang, Xuan Yang, Yandong Li, Tommi Jaakkola, Xuhui Jia, and Saining Xie. Inference-time scaling for diffusion models beyond scaling denoising steps. *arXiv preprint arXiv:2501.09732*, 2025.

[36] Xinyin Ma, Gongfan Fang, and Xinchao Wang. Deepcache: Accelerating diffusion models for free. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 15762–15772, 2024.

[37] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, June 1970.

[38] John C. McCallum. Disk drive prices (1955–2023). https://jcmit.net/diskprice.htm, 2023. Monthly survey of consumer HDD prices from NewEgg.com, 1955–2023. Archived at https://web.archive.org/web/2024/https://jcmit.net/diskprice.htm. Accessed 2026-04.

[39] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumber, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, 2018.

[40] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. f4: Facebook's warm BLOB storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 383–398, 2014.

[41] Shadi A Noghabi, Sriram Subramanian, Priyesh Narayanan, Sivabalan Narayanan, Gopalakrishna Holla, Mammad Zadeh, Tianwei Li, Indranil Gupta, and Roy H Campbell. Ambry: Linkedin's scalable geo-distributed object store. In *Proceedings of the 2016 International Conference on Management of Data*, pages 253–265, 2016.

[42] NVIDIA. CUDA graphs. https://developer.nvidia.com/blog/cuda-graphs/, 2019.

[43] NVIDIA. TensorRT: High-performance deep learning inference sdk. https://developer.nvidia.com/tensorrt, 2025. Accessed: 2026-04.

[44] PCPartPicker. Price trends: GeForce RTX 4090. https://pcpartpicker.com/trends/price/video-card/#gpu.chipset.geforce-rtx-4090, 2026. Accessed: 2026-04.

[45] PCPartPicker. Price trends: GeForce RTX 5090. https://pcpartpicker.com/trends/price/video-card/#gpu.chipset.geforce-rtx-5090, 2026. Accessed: 2026-04.

[46] William Peebles and Saining Xie. Scalable diffusion models with transformers. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 4195–4205, 2023.

[47] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR, 2021.

[48] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.

[49] Robi Rahman. Performance per dollar improves around 30% each year. https://epoch.ai/data-insights/price-performance-hardware, 2024. Epoch AI data insight. Accessed 2026-04.

[50] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125*, 1(2):3, 2022.

[51] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. In *International conference on machine learning*, pages 1278–1286. PMLR, 2014.

[52] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10684–10695, 2022.

[53] Jiaming Song, Chenlin Meng, and Stefano Ermon. Denoising diffusion implicit models. In *International Conference on Learning Representations (ICLR)*, 2021.

[54] Irina C. Tuduce and Thomas Gross. Adaptive main memory compression. In *USENIX Annual Technical Conference (ATC)*, pages 237–250, 2005.

[55] Aaron Van Den Oord, Oriol Vinyals, et al. Neural discrete representation learning. *Advances in neural information processing systems*, 30, 2017.

[56] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, Santa Clara, CA, February 2015. USENIX Association.

[57] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. TMO: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 609–621, 2022.

[58] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The case for compressed caching in virtual memory systems. In *USENIX Annual Technical Conference (ATC)*, pages 101–116, 1999.

[59] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, and Andrew Warfield. Characterizing storage workloads with counter stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 335–349, Broomfield, CO, October 2014. USENIX Association.

[60] Xingbo Wu, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, and Song Jiang. zexpander: a key-value cache with both high performance and fewer misses. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[61] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. Fifo queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 130–149, New York, NY, USA, 2023. Association for Computing Machinery.

[62] Tianwei Yin, Michaël Gharbi, Richard Zhang, Eli Shechtman, Fredo Durand, William T Freeman, and Taesung Park. One-step diffusion with distribution matching distillation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 6613–6623, 2024.

[63] Xingyu Zheng, Xianglong Liu, Yichen Bian, Xudong Ma, Yulun Zhang, Jiakai Wang, Jinyang Guo, and Haotong Qin. Bidm: Pushing the limit of quantization for diffusion models. *Advances in Neural Information Processing Systems*, 37:39009–39035, 2024.

[64] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenjie Liu, and Tianming Yang. Demystifying cache policies for photo stores at scale: A tencent case study. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 284–294, 2018.