Similarity Caching for Text-to-Image Diffusion Models

Internship Report

Pietro Tellarini
Supervisors: Giovanni Neglia, Sara Alouf
MSc Computer Science — UbiNet Track

August 24, 2025

Abstract

Text-to-image diffusion models deliver impressive visual quality but incur high compute and latency, which complicates large-scale, real-time serving. We study *similarity caching* reusing a previously generated image when a new prompt is sufficiently close in an embedding space as a systems lever to reduce end-to-end cost. We formalize the problem with a cost model that normalizes the approximation cost around an acceptance threshold θ, and we build a reproducible pipeline that (i) embeds prompts with CLIP, (ii) performs nearest-neighbor lookup via FAISS, and (iii) evaluates online cache policies. Beyond thresholded LRU/LFU baselines, we implement two policies tailored to similarity reuse: *qLRU*-ΔC (probabilistic admission/refresh proportional to saved cost) and *Duel* (pairwise challenger–incumbent replacement by accumulated gains). On DiffusionDB prompt traces, across capacities and thresholds.

Contents

1 Introduction **3**
1.1 Motivation . . . 3
1.2 Objectives and Contributions . . . 3

2 Background and Context **4**
2.1 Similarity Caching . . . 4
2.2 Diffusion Models . . . 5
2.3 Related Works on Similarity Caching . . . 6

3 Materials and Methods: Technology and Design Choices **6**
3.1 Representations and Similarity . . . 7
3.2 Vector Search Backbone . . . 7
3.3 Policies Evaluated . . . 7
3.4 Cost normalization and acceptance rule . . . 9
3.5 Dataset and Preparation Pipeline . . . 10
3.6 Code Architecture & Implementation . . . 11
3.7 Benchmark driver and parameter grids . . . 12
3.8 Metrics . . . 12

4 Results **13**
4.1 Exploratory Analysis: Users, Sessions, and Correlations . . . 13
4.2 Results: overall patterns across thresholds and capacities . . . 13
4.3 Reproducibility . . . 15

5 Conclusions **16**

1 Introduction

In recent years, artificial intelligence has achieved remarkable progress in the field of *text-to-image generation*. This technology allows a user to describe a scene or an object in natural language and automatically obtain a synthetic image that matches the description. Popular tools such as *DALL-E* [1, 2], *Stable Diffusion* [3], or *MidJourney* [4] have demonstrated how powerful and accessible these models can be, opening new opportunities for creativity, design, and communication.

However, these systems are computationally very demanding: generating a single image often requires powerful GPUs and several seconds of processing. This makes it challenging to use text-to-image models at large scale, for example when serving thousands of requests in real time.

To address this issue, researchers have explored different acceleration techniques. One of the most promising is *similarity caching*, an idea borrowed from computer systems: if two user requests are very similar, it may be possible to reuse a result that has already been computed instead of running the whole generation process again. This approach can save time and resources, provided that the reused result is still of sufficient quality.

In this report, I investigate how similarity caching can be applied to diffusion models for text-to-image generation. The work was carried out during my internship with the NEO team at Inria, and it combines concepts from machine learning, information retrieval, and systems design. The following sections introduce the motivation, the objectives, and the main contributions of the project.

1.1 Motivation

Text-to-image diffusion models deliver striking visual quality but require tens of iterative denoising steps per request, making end-to-end latency and GPU-hours the dominant cost in production. Real prompt streams, however, exhibit strong locality (same user, same session, near-duplicates or small edits), which opens the door to *similarity caching*: reuse a previously generated output when a new prompt is sufficiently close in an embedding space. Compared to exact caching, similarity caching reasons about *approximate hits* and their *quality* versus the full regeneration cost, turning the problem into a principled cost–benefit trade-off. In this project we build an end-to-end pipeline and we study *online* cache policies that do not rely on stationary popularity estimates, which are unrealistic on short, non-stationary traces.

1.2 Objectives and Contributions

The main goal of this internship project is to design, implement, and evaluate strategies to reduce the computational cost of text-to-image diffusion models through *similarity caching*. In simple terms, the idea is to avoid generating an image from scratch if a very similar request has already been processed in the past. Instead, the system can try to reuse the previous result, provided that the quality remains acceptable. More concretely, the contributions of this work are:

1. **Design and implementation of cache policies.** In computer systems, a *cache policy* determines which results are stored for reuse and when they should be replaced. In this project, we implemented and adapted policies that can make decisions without relying on long-term statistics about request frequencies (often called "λ-unaware" policies). This makes them suitable for real usage scenarios, where user requests are diverse and constantly changing. For comparison, we also implemented simple baseline policies (such as Least Recently Used or Least Frequently Used) adapted to the similarity setting.

2. **Creation of a reproducible experimental framework.** We built a complete pipeline that takes text prompts from a public dataset, converts them into numerical embeddings using the CLIP model, and then uses a vector-search library (FAISS) to quickly find similar

requests. On top of this backbone, we developed a simulation environment where different cache policies can be tested under realistic workloads.

3. **Analysis tools and results.** We produced scripts and dashboards to visualise and analyse the trade-offs between latency (time saved) and quality (similarity of reused results). These tools make it possible to compare policies, tune their parameters, and identify which configurations are most effective under different requirements.

Overall, this work provides both a methodological framework and practical insights into how similarity caching can make text-to-image diffusion models more efficient. It highlights when reuse is safe, how different policies affect the balance between speed and quality, and how these choices translate into measurable savings in GPU time.

2 Background and Context

Caching is a classical technique in computer systems: when a resource is requested multiple times, storing it temporarily allows future requests to be served faster and at lower cost. Traditional caches usually work with exact matches: the same file, the same image, or the same data item must be requested again in order to benefit from reuse.

In the context of text-to-image generation, however, user requests are often similar but not identical. For example, a user may first ask for *"a cat sitting on a sofa"* and then for *"a small kitten sitting on a sofa"*. While the prompts are not exactly the same, they are close enough that the second result could potentially be approximated by reusing the first one. This idea leads to *similarity caching*: instead of requiring exact equality, the system checks whether a new request is sufficiently close to something already stored in the cache, according to a similarity measure.

2.1 Similarity Caching

Cost model. We adopt the similarity-caching framework introduced by Neglia *et al.*, which extends exact caching to settings where requests can be served approximately by *similar* objects [5]. In our setting, each incoming request is denoted by x (for example, a text prompt). The cache state at a given time is represented by S, i.e. the set of items currently stored, and after an update it may change to a new state T. Two types of costs are defined:

- **Exact computation:** running the full diffusion process, with a fixed cost $C_r > 0$ (high latency, high GPU time).
- **Approximate reuse:** finding a similar item y already in the cache and reusing it, with an *approximation cost* $C_a(x, y)$ that depends on how close the two requests are. If no sufficiently similar item is found, $C_a(x, S) = +\infty$.

The service cost of a request x given a cache state S is therefore:

$$C(x, S) = \min\{C_a(x, S),\ C_r\} \tag{1}$$

meaning that the system chooses the cheaper option between reusing a similar item (if good enough) and computing a fresh result.

Cache updates. When inserting new items, the cache can only replace one element at a time. If the cache state changes from S to T, the *movement cost* is:

$$C_m(T, S) = \begin{cases} 0 & \text{if } S = T \text{ (no change)} \\ C_r & \text{if one object is replaced} \\ +\infty & \text{if more than one object changes at once.} \end{cases} \tag{2}$$

Why this model matters. This formalisation allows us to compare different caching strategies in a principled way. By looking at the average cost across a sequence of requests, we can measure whether a policy achieves real savings in latency and GPU usage, while still providing results of acceptable quality. We can define a cost function over a request sequence r_T, the average cost of a policy A is

$$C_{\mathcal{A}}(S_1, r_T) \;=\; \frac{1}{T}\sum_{t=1}^{T}\big[\, C_m(S_t, S_{t+1}) \;+\; C(r_t, S_{t+1})\,\big]. \tag{3}$$

λ-unaware policies. In similarity caching, if request x arrives with rate λ_x, the expected cost of a fixed cache state S is $C(S) = \sum_x \lambda_x\, C(x, S)$. λ-*aware* policies try to minimise this quantity by estimating the rates $\{\lambda_x\}$, which only makes sense when popularities are sufficiently stationary and can be learned reliably. In our setting the traces are finite and non-stationary, and objects live in a continuous embedding space, so accurate rate estimates are not available. We therefore use λ-*unaware* online policies (qLRU-ΔC, Duel) that rely only on instantaneous costs (approximation vs. retrieval) and local decisions, together with thresholded LRU/LFU baselines; this keeps the evaluation robust to popularity drift without assuming a model we cannot validate.

2.2 Diffusion Models

Diffusion models are a family of generative models that have recently achieved state-of-the-art results in image synthesis [6, 7, 8]. The idea is to start from random noise and progressively remove it through a sequence of denoising steps, guided by a neural network. When the process is conditioned on text, the result is a *text-to-image* model: the input is a written prompt and the output is an image matching the description.

In the following, we recall the main equations used to formalise the *forward process*, which gradually adds noise to clean data, and a *reverse process*, where a neural network learns to remove the noise step by step until a new realistic sample is produced (fig 1).

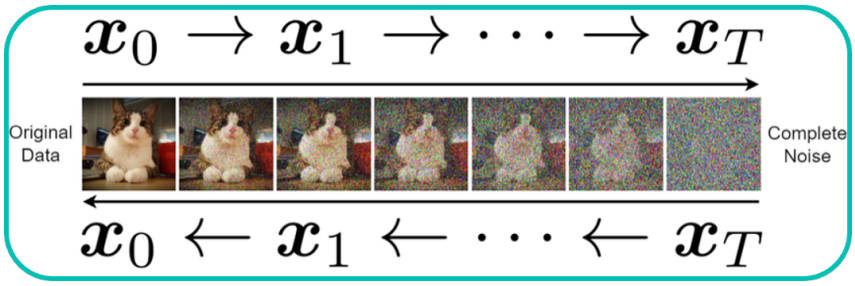


Figure 1: on the top we start from a clean image x_0, progressively corrupted into x_t until reaching pure noise x_T; in the bottom the reverse denoising trajectory, where the trained model reconstructs a new realistic image step by step.

Notation. We denote by x_0 the original clean data (for example, an image). At each timestep $t = 1, \ldots, T$, the forward process produces a noisier version x_t, until after T steps we obtain x_T, which is almost pure Gaussian noise. The reverse process then starts from x_T and iteratively denoises it, eventually recovering a new sample that resembles data from the training distribution.

Forward process. We denote by x_0 the original clean image. At each step $t = 1, \ldots, T$, Gaussian noise is added to obtain a progressively more corrupted version x_t, until the final step x_T is essentially pure noise. This is formalised as:

$$q(x_t \mid x_{t-1}) = \mathcal{N}\Big(\sqrt{1-\beta_t}\, x_{t-1},\ \beta_t I\Big)\,,$$

where β_t controls the amount of noise added at step t. Thanks to a closed-form derivation, we can also directly sample x_t from x_0.

Reverse process. The generative model is trained to invert this corruption process. A neural network parameterised by θ predicts the noise component added at each step and thus learns a conditional distribution

$$p_\theta(x_{t-1} \mid x_t) = \mathcal{N}\big(\mu_\theta(x_t, t), \Sigma_t\big).$$

Training typically uses the simplified objective proposed by Ho and Salimans in *Classifier-Free Guidance* [9], which improves sample quality by directly predicting the noise ϵ added at each step.

Conditional generation. When the process is conditioned on text, we obtain a *text-to-image diffusion model*. In this case, the network receives both the noisy input x_t and a text embedding c derived from the prompt. Techniques such as classifier-free guidance [9] adjust the balance between fidelity to the prompt and diversity of the generated samples.

2.3 Related Works on Similarity Caching

In practice, the approach most closely related to ours is *GPTCache* developed by Pinecone/Zilliz [10]. GPTCache implements a semantic cache for large language models: requests are embedded into a vector space, and approximate matches are retrieved through similarity search using vector databases. Although focused on natural language rather than images, this design principle is directly aligned with our approach: store embeddings of past requests, and serve approximate hits when new inputs fall sufficiently close in embedding space.

A complementary line of work is represented by *NIRVANA* [11], which accelerates text-to-image diffusion models by reusing intermediate denoising states instead of final outputs. While promising, this approach differs fundamentally from ours: it requires modifications inside the generative pipeline, whereas our method treats the diffusion model as a black box and focuses on semantic-level reuse of completed generations.

Taken together, these works highlight a growing interest in similarity-based acceleration techniques across domains. Our contribution is to bring the semantic caching paradigm, already validated in the LLM space, to diffusion models and to provide a reproducible framework for systematically evaluating different cache policies under realistic workloads.

3 Materials and Methods: Technology and Design Choices

System Overview

At a high level, the system works as follows. Every time a user submits a text prompt, it is first converted into a numerical representation (embedding) using the CLIP model. This embedding is then compared against the embeddings stored in the cache:

- If a sufficiently similar prompt is already in the cache, the system reuses the corresponding generated image (an *approximate hit*). This saves GPU time, since no new diffusion run is needed.
- If no similar prompt is found, the system generates a fresh image by running the full diffusion model (a *miss*). The new prompt and its embedding are then inserted into the cache, possibly replacing an older entry according to the chosen cache policy.

In this way, the cache acts as a memory of all previously seen prompts and their results. The core design questions are: (i) how similarity is defined (via embeddings and thresholds), (ii) how

cache entries are managed (insertion and eviction policies), and (iii) how the trade-off between reuse quality and computational savings is quantified. The following subsections describe these components in detail.

3.1 Representations and Similarity

To decide whether two prompts are similar, we need a way to translate text into a numerical form that can be compared efficiently. For this purpose we use *CLIP* (Contrastive Language–Image Pre-training) [12], a model jointly trained on text–image pairs to align their representations in a shared embedding space. In simple terms, it learns to produce vectors (embeddings) such that a caption like "a cat on a sofa" is close to the embedding of an image of a cat on a sofa, while unrelated text or images are far apart.

This property makes CLIP ideal for our use case: by embedding each prompt into a vector and normalising it, we can measure similarity between prompts simply as the cosine similarity between their embeddings. A high similarity score means the two prompts are semantically close, and thus a candidate for reuse in the cache. We ℓ_2-normalize once at creation time, in this way cosine similarity reduces to an inner product on normalized vectors,

$$\mathrm{sim}(u, v) \;=\; \frac{u^\top v}{\|u\|\,\|v\|} \;=\; u^\top v\,,$$

which serves two roles in our stack: (i) it quantifies the *quality* of approximate hits; (ii) it implements the acceptance rule for similarity hits via a threshold θ (serve from cache if $\mathrm{sim} \geq \theta$).

3.2 Vector Search Backbone

Once we have vector representations of prompts, we need to efficiently search among thousands of embeddings to find the most similar ones. For this task we rely on *FAISS* (Facebook AI Similarity Search) [13], a library optimised for large-scale nearest-neighbour search. FAISS is widely used in industry because it can handle billions of vectors and supports GPU acceleration for fast queries.

In our system, FAISS provides the backbone for cache lookup: given a new prompt embedding, it finds the closest candidates among previously stored embeddings. We use exact inner-product search on ℓ_2-normalised vectors, which allows cosine similarity to be computed as a simple dot product. This guarantees reproducible results and keeps the system efficient enough to run large-scale benchmarks.

3.3 Policies Evaluated

We restrict to λ-unaware policies tailored to similarity caching (no reliance on stationary popularity estimates), plus simple baselines adapted to similarity.

Threshold LRU and LFU. As a reference point, we implemented two classical cache replacement policies: Least Recently Used (LRU) and Least Frequently Used (LFU). These are considered baselines because they are simple, widely adopted in computer systems, and do not require prior knowledge of request statistics.

In LRU, the cache evicts the item that has not been accessed for the longest time, while in LFU it evicts the item with the lowest request frequency. Both are easy to implement and capture intuitive notions of temporal and popularity locality.

In our similarity-caching setting, these baselines are adapted with a *threshold-based rule*: given a new request x, we compute its embedding and compare it with all cached embeddings. If the similarity with the best candidate is above a predefined threshold θ, the request is considered

an *approximate hit*. In this case, the corresponding cache entry is reused and its recency (for LRU) or frequency counter (for LFU) is updated.

If no candidate exceeds the threshold, the request is a *miss*. The system then recomputes the result from scratch using the full diffusion model. The newly generated image and its prompt embedding are then inserted into the cache: *what* is stored are both the embedding (for future lookups) and the generated image (for reuse), and *where* it is placed depends on the chosen replacement policy (the least recently used item is evicted in LRU, or the least frequently used one in LFU).

qLRU-ΔC (online, λ-unaware). The idea of qLRU–ΔC is to maintains an LRU queue, while making cache updates probabilistic, based on the *gain in cost* provided by a candidate object. Instead of always inserting or refreshing deterministically, the policy assigns a probability to these actions: objects that save more cost are more likely to be admitted or refreshed, while marginal ones are ignored.

Formally, let $z = \arg\min_{y \in S} C_a(x, y)$ be the best approximator of request x among the items in cache S. Here C_a denotes the approximation cost as defined in the similarity model (Sec. 2.1). Let $C(x, S \setminus \{z\})$ be the service cost if z were absent, i.e. either using the next-best candidate in the cache or falling back to recomputation at cost C_r. The qLRU policy then assigns probabilities based on these quantities:

$$p_{\text{refresh}} = \frac{C(x, S \setminus \{z\}) - C_a(x, z)}{C_r},$$

the cache entry z is *moved to the front*, as in standard LRU. Intuitively, objects that yield larger cost savings are refreshed more often, increasing their chance of staying longer in the cache.

$$p_{\text{insert}} = q\,\frac{C_a(x, z)}{C_r},$$

a new entry for the exact request $x$is inserted at the front of the cache, possibly evicting the tail item. The factor q is the only hyper-parameter of the policy: $q = 1$ recovers classical LRU, while smaller q reduces insertion frequency and leads to more stable cache contents.

Input: request x, state S_t (ordered), parameters $q \in (0, 1]$, C_r
if $C_a(x, S_t) > C_r$ **then**
  retrieve x; with prob. q insert x at front; evict from tail if needed
else
  $z \leftarrow \arg\min_{y \in S_t} C_a(x, y)$; serve z
  with prob. $\dfrac{C(x, S_t \setminus \{z\}) - C_a(x, z)}{C_r}$ move z to front
  with prob. $q\,\dfrac{C_a(x, z)}{C_r}$ retrieve x and insert at front (evict if needed)
end

Algorithm 1: qLRU-ΔC pseudocode

Duel Cache (online, λ-unaware). When a new request $y^\star \notin S_t$ arrives, the algorithm starts a "duel" between an incumbent $y \in S_t$ and the challenger $y^\star$. For subsequent requests, whichever of the two provides the better approximation accrues a gain $\Delta(\cdot)$ equal to the cost reduction it achieves. The incumbent y is replaced by $y^\star$ if the challenger's cumulative advantage exceeds a margin δ within a timeout τ; otherwise, the incumbent is kept.

The choice of which incumbent y to duel against is stochastic: with probability β, the algorithm selects the closest incumbent (local matching), while with probability $1-\beta$ it selects a random incumbent uniformly (global matching).

Incremental gains. Following the service cost (1) of request r under cache state S during a duel between incumbent $y \in S$ and challenger $y^\star$, the counters are updated as

$$\Delta_y(r;S) \;=\; C(r,\, S\setminus\{y\}) - Ca(r,y), \qquad \Delta_{y^\star}(r;S) \;=\; C(r,\, S) - Ca(r,y^\star).$$

Thus, Δ represents the incremental cost reduction that each duellist achieves relative to serving r without it.

Input: Cache size k, parameters δ, τ, β
Output: Cache state evolution $\{S_t\}$
for *each request* r_t **do**
  if $r_t \in S_t$ **then**
    Serve r_t (exact hit)
  else
    Let $y^\star \leftarrow r_t$ (challenger)
    Select incumbent $y \in S_t$:
      with probability β: y is closest to $y^\star$
      with probability $1-\beta$: y is chosen uniformly at random
    Start duel between $(y, y^\star)$ with counters $c(y)=0$, $c(y^\star)=0$
    while *duel not terminated* **do**
      Upon future request $r_{\tilde{t}}$: **if** y *is best approximator for* $r_{\tilde{t}}$ **then**
        $c(y) \leftarrow c(y) + \Delta(r_{\tilde{t}}, y)$
      end
      if $y^\star$ *is best approximator for* $r_{\tilde{t}}$ **then**
        $c(y^\star) \leftarrow c(y^\star) + \Delta(r_{\tilde{t}}, y^\star)$
      end
      if $|c(y^\star) - c(y)| > \delta$ *or time* $> \tau$ **then**
        terminate duel
      end
    end
    if $c(y^\star) > c(y) + \delta$ **then**
      Replace y with $y^\star$ in S_t
    end
    else
      Discard $y^\star$ (keep y)
    end
  end
end

Algorithm 2: Duel Cache pseudocode

3.4 Cost normalization and acceptance rule

We adopt the similarity-caching framework of Eq.1 - 3: the service cost is $C(x,S) = \min\{C_a(x,S),\, C_r\}$ and a single replacement costs $C_m = C_r$. In all experiments we fix the regeneration cost to unit value

$$\boxed{C_r = 1}$$

and we define an approximation cost *centered* at the acceptance threshold $\theta \in [0,1)$.

Let $u, v \in \mathbb{R}^d$ be ℓ_2-normalized CLIP embeddings and let $s(u, v) = \cos(u, v) \in [-1, 1]$ denote their cosine similarity. In the general framework the approximation cost $C_a(\cdot, \cdot)$ is left abstract: any application-specific distance or distortion measure can be used.

In our setting we adopt the cosine-threshold rule: given a cache state S, a request x can be served approximately from $z \in S$ iff $s(x, z) \geq \theta$. To cast this rule into the cost framework (*accept iff* $C_a \leq C_r$), we define the following application-specific cost map:

$$\boxed{C_a^{(\theta)}(u, v) = \frac{1 - s(u, v)}{1 - \theta}} \tag{4}$$

This mapping has two immediate properties:

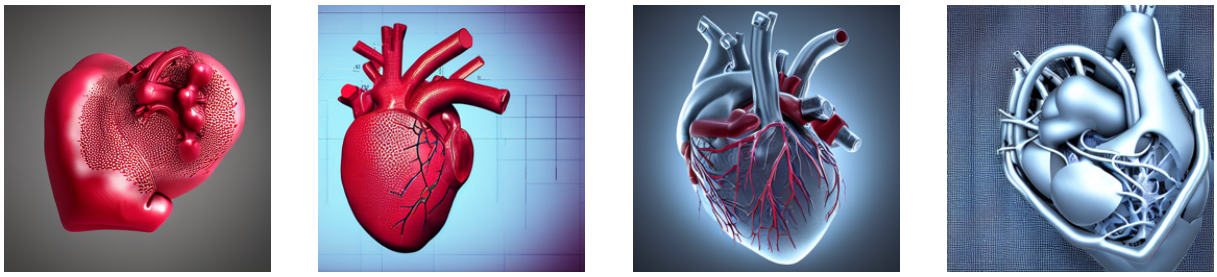
- $C_a^{(\theta)}(u, v) \leq 1 \iff s(u, v) \geq \theta$, i.e. the cosine-threshold acceptance is exactly equivalent to $C_a \leq C_r$ with $C_r = 1$;
- conditionally on acceptance ($s \geq \theta$), $C_a^{(\theta)} \in [0, 1]$ with $C_a^{(\theta)} = 0$ at perfect similarity ($s = 1$) and $C_a^{(\theta)} = 1$ at the threshold ($s = \theta$).

Hence all increments or probabilities used by policies such as qLRU-ΔC or Duel are normalized in $[0, 1]$ for any choice of θ, making results directly comparable across thresholds while keeping the general framework of [5] unchanged.

3.5 Dataset and Preparation Pipeline

DiffusionDB Our analysis relies on *DiffusionDB*, the first large-scale community dataset of text–to–image generations with Stable Diffusion [14, 15, 16]. The full release contains over 14 million images (about 6.5 TB) paired with 1.8 million unique prompts, collected from real user interactions on public Discord servers. Each generation is accompanied by a JSON metadata entry that records the conditioning `prompt`, the hyperparameters used (e.g., sampler, guidance scale, number of steps), a `timestamp`, and an optional anonymised `user` identifier.

For practical reasons, we build on the *DiffusionDB small metadata* release, which provides the same JSON records without the corresponding image files. This allows us to process prompt distributions and hyperparameter statistics at scale. In addition, whenever needed for illustration or validation, we recover a subset of actual generations from the ∼1.65 TB *DiffusionDB 2M* collection, which contains 2 million images and approximately 1.5 million prompts in the same format as the large dataset 2.



Original image | similarity 0.9555 | similarity 0.8694 | similarity 0.8694

Figure 2: Images from DiffusionDB: three nearest neighbors retrieved by cosine similarity of textual CLIP embeddings, starting from a random image.

Cleaning and embedding preparation We curate DiffusionDB into a clean Parquet table by keeping only `prompt`, `timestamp`, and (when available) `user_name`. Timestamps are normalised to UTC and stored as `datetime64`; we drop empty/null prompts and apply light cleaning (trim/collapse whitespace, strip non-printable characters).

We then precompute CLIP text embeddings offline on two Nvidia Tesla T4 GPUs (mixed precision for throughput). Each vector is ℓ_2-normalised and stored as `float32`. The resulting `clip_emb` column is saved back into Parquet filetype, so downstream runs load a $N \times d$ matrix of precomputed embeddings. The output is a single Parquet dataset with this *effective* schema:

Column	Meaning
`prompt`	Text prompt (UTF–8, lightly cleaned)
`timestamp`	UTC time the prompt was issued (`datetime64`)
`user_name`	Optional user identifier (may be missing)
`clip_emb`	CLIP text embedding, length d (`float32` list)

3.6 Code Architecture & Implementation

The stack separates the dataset layer, the cache policy layer (concrete policies inherit from a single base), and the simulator driver. Backend utilities (FAISS, thresholds) are composed into the base and policies. The code is availabe on the GitHub repository.

Main Modules.

- `PromptDatasetManager.py` — loads a Parquet file with columns `[prompt, clip_emb, user_name, timestamp]`, builds a contiguous $N \times d$ float32 embedding matrix, normalises timestamps, and exposes sequential, random, and sessioned sampling utilities (per-user segments cut by a time gap). It also precomputes a global time order for sequential replays.
- `BaseCache.py` — All policies subclass BaseSimilarityCache (FAISS index ops, threshold rule, observer hooks), then implement their own query() semantics. CacheSimulator replays a (keys, embeddings, indices) triplet through any policy, logs an event stream (hit/miss/add/evict with similarity), and returns a per-run summary (hit rate, avg similarity, duration). The simulator powers both CLI benchmarks and the live dashboard.
  - `CachePolicy.py` — LRU/LFU/TTL (thresholded for similarity).
  - `CacheUnAware.py` — qLRU-ΔC / Duel.
- `Backend.py` — FAISS/back-end helpers.
- `Dashboard.py` - (Panel/HoloViews) lets you watch hit-rate and hit-quality evolve in real time for a chosen policy, capacity, threshold, and trace type

Relationships. Concrete caches inherit from `BaseSimilarityCache`; `PromptDatasetManager` feeds the simulator; the simulator drives policy calls; back-end utilities are composed where needed (NN queries, thresholds).

Extensibility. To add a new policy, subclass `BaseSimilarityCache`, implement `query()` and update hooks, and register it in the benchmark CLI grid.

3.7 Benchmark driver and parameter grids

Outputs and manifest. All experiments are launched via `benchmark_cache_policies.py`, which: (i) loads the Parquet dataset through the manager; (ii) builds unique keys and a trace index; (iii) expands a policy grid; (iv) runs each configuration; (v) writes a run-level `summary.csv` and per-run histories in `histories/` (event stream with hit/miss/add/evict and similarity). For each run we record `policy`, hyper-parameters, hit/miss counts, hit rate, average similarity, the overall policy $C_{\mathcal{A}}$ cost as defined in 3 and wall-clock duration; the job manifest includes creation time, number of runs/requests, CLI args, and paths to `summary.csv` and `histories/`.

Parameter grids. The following tables summarize the experimental settings. Table 1 reports the global grid and fixed values shared across all runs, while Table 2 details the hyperparameters specific to each policy. The policy were run on a subset of the dataset of 20000 requests.

Table 1: Global grid and fixed settings used in all runs.

Item	**Values / Notes**
Capacity (% of total Reqeusts)	{0.1, 1.0, 5.0, 10.0}
Threshold θ	{0.65, 0.75, 0.85}
Regeneration cost C_r	1 (fixed; unit of cost)
Approx. cost $C_a^{(\theta)}(s)$	$(1-s)/(1-\theta)$
Acceptance rule	$C_a^{(\theta)} \leq C_r$ (i.e., $\cos \geq \theta$)

Table 2: Policy-specific hyper-parameter grid (values swept unless marked as fixed).

Policy	**Parameter**	**Symbol**	**Values / Fixed**
LRU / LFU	(thresholded baselines)	–	uses Table 1
qLRU-ΔC	Insert factor (exact admission)	q	{0.1, 0.2, 0.4}
Duel	NN-bias (pick incumbent as NN)	β	{0.6, 0.75}
	Margin on cumulative savings	δ	{0.02, 0.05}
	Horizon (requests)	τ	{100, 200}
	Neighbors per request	–	8 (fixed)
	Max active duels	–	8 (fixed)

3.8 Metrics

We report three primary metrics, computed on the evaluation stream and then aggregated *per prompt* to generate the figures:

- **Hit Rate (HR)**: fraction of requests for which the policy serves an approximate hit (equivalently, $\cos \geq \theta$).
- **Quality (prompt)**: for each prompt p, the mean cosine similarity over its *accepted* hits; we then aggregate these values across prompts (mean ± CI) or show them directly in the scatter plots.
- **Total cost** $C_{\mathcal{A}}$: the similarity caching objective (3), with regeneration cost fixed to C_r=1, C_m=C_r only on replacements, and the post-state service cost (1) is

$$C(r_t, S_{t+1}) = \min\Big\{ C_a^{(\theta)}(s_t),\, 1 \Big\},$$

with s_t the top-1 cosine similarity measured *after* the policy's update at time t, and $C_a^{(\theta)}$ derived from 4.

GPU-grounded wall-clock. Our primary objective remains the theoretical cost $C_{\mathcal{A}}$ (Sec. 2.1), which is dimensionless by construction (C_r=1). When we want an interpretable wall-clock proxy on a given GPU we estimate the time as the sum of full regenerations (misses), because lookups/refreshes are negligible compared to a diffusion run:

$$\text{time}_{\text{total}} \;\approx\; N_{\text{miss}} \cdot \tau_{\text{img}} + N_{\text{hits}} \cdot t_{\text{pipeline}}$$

For reproducibility, we ground τ_{img} on public SD 1.5 (512×512) inference benchmarks on an RTX 4090, which report about **75 images/minute** (i.e., $\tau_{\text{img}} \approx 0.8\,\text{s}$ per image) [17]. We therefore use τ_{img}=0.8 s as default. This mapping yields "wall-clock seconds" for our plots without changing the theoretical metric. Following the same idea we estimated a t_{pipeline} of 0.02 second per hit. This overall time needs just to give an approximate run time measure for the reader.

4 Results

4.1 Exploratory Analysis: Users, Sessions, and Correlations

Before benchmarking, we explore prompt dynamics to tune meaningful cache capacities and thresholds.

We analyse the curated DiffusionDB slice used in our experiments. After cleaning and deduplication (Sec. 3.5), we analyze prompts with UTC timestamps and precomputed CLIP embeddings (dim 512). Prompts exhibit clear temporal/user locality: the cosine to the previous prompt of the same user is markedly higher than to a random prompt, and even higher when considering the sessions, suggesting that small caches can already capture within-session reuse 3. Session gaps show a strict correlation based on time, smaller is the gap, more similarity occours between prompts 4. Another important observation is given by the histogram of reuse distances 5 where we can notice how the similar request are close in terms of time above the threshold of 0.8.

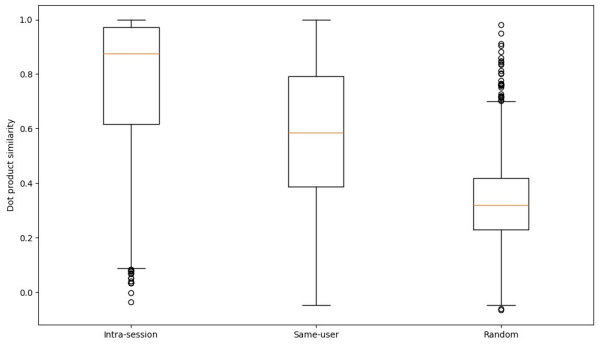


Figure 3: Mean similarity based on different gaps from the datasets. This highlight time correlation between prompts of same users.

4.2 Results: overall patterns across thresholds and capacities

Reading guide. Each panel fixes the cache capacity and the acceptance threshold θ (we facet by θ) 6; markers encode policies and hyper-parameters. The x axis is *Quality* (mean cosine on accepted hits). Depending on the figure set, the y axis is either *Hit Rate* (HR) or the theoretical cost $C_{\mathcal{A}}$ (movement + post-state service with C_r=1 and $C_a^{(\theta)}(s) = \frac{1-s}{1-\theta}$; lower is better). Legends also include an RTX 4090 wall-clock proxy for interpretability (Sec. 3.8).

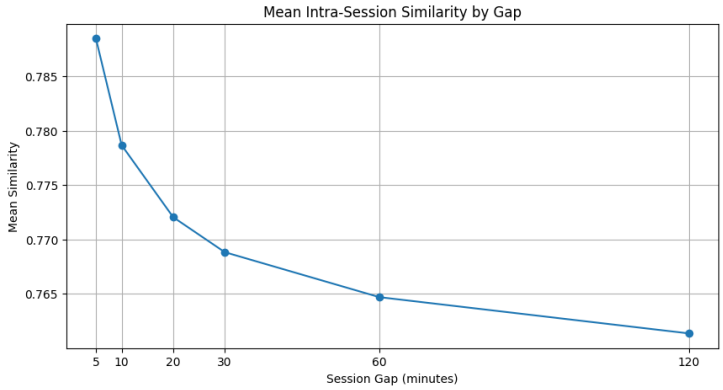


Figure 4: Mean similarity based on different gaps from the datasets. This highlight time correlation between prompts of same users.

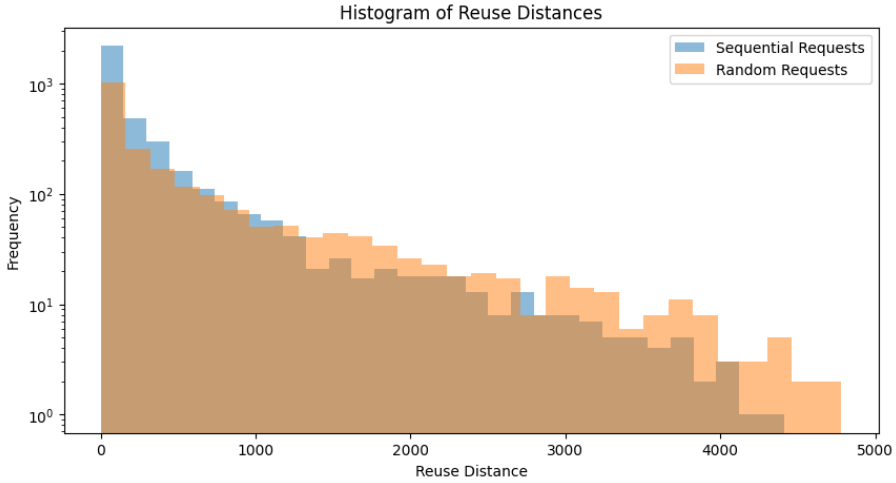


Figure 5: Reuse distance between similar prompt ($\theta = 0.8$) depending on consecutive requests order.

Effect of the threshold. Raising θ from 0.65 to 0.85 shifts points to the right (higher measured quality by construction) and down (HR) or up ($C_{\mathcal{A}}$): at cap 200, LRU's HR drops by $\sim$0.20–$\sim$0.25 (e.g., $\sim$0.73 $\rightarrow$ $\sim$0.50) while quality increases modestly; all other policies follow the same trend. This reflects stricter acceptance: approximate hits become rarer and more expensive.

Effect of capacity. Increasing capacity from 20 to 2000 improves both HR and Quality and lowers $C_{\mathcal{A}}$ for every policy. Gains are largest from 20$\rightarrow$200 (e.g., at θ=0.65, LRU's HR rises from $\sim$0.66 to $\sim$0.73) and taper off thereafter. In the cost plots, points move downward markedly (e.g., θ=0.65: LRU $C_{\mathcal{A}} \approx 0.679$ at cap 200 and ≈ 0.608 at cap 2000; qLRU-ΔC with q=0.4 drops from ≈ 0.653 to ≈ 0.540).

Policy ordering (consistent across panels).

- **LRU** is the most reliable frontier policy. It consistently achieves the highest HR at a given quality and the lowest $C_{\mathcal{A}}$ at small/medium capacity and/or high θ. Examples: cap 20, θ=0.85: $C_{\mathcal{A}} \approx 0.738$ at quality $\sim$0.76 while Duel/LFU are $>$0.95 and qLRU-ΔC $\geq$0.80; cap 200, θ=0.75: $C_{\mathcal{A}} \approx 0.682$ at quality $\sim$0.77.
- **qLRU-ΔC** forms a monotone family as q increases. The setting q=0.4 sits near the frontier in *every* panel and becomes *cost-optimal* at low θ and large caches. Representative points

Table 3: Policies on (or closest to) the Pareto frontier in (Quality, $C_{\mathcal{A}}$) across thresholds (examples for cap=200 and cap=2000).

θ	cap=200	cap=1000	cap=2000
0.65	LRU, qLRU-ΔC (q=0.4)	LRU $\approx$ qLRU-ΔC (q=0.4)	qLRU-ΔC (q=0.4), LRU
0.75	LRU (frontier), qLRU-ΔC (q=0.4) near	LRU (frontier)	LRU (frontier)
0.85	LRU (frontier)	LRU (frontier)	LRU (frontier)

at θ=0.65: cap 200: $C_{\mathcal{A}} \approx 0.653$ (quality $\sim$0.75) vs LRU 0.679; cap 1000: 0.592 vs LRU 0.655; cap 2000: 0.540 vs LRU 0.608. Lower q values (0.2, 0.1) are dominated (lower quality and higher cost).

- **Duel** is below LRU/qLRU on HR and $C_{\mathcal{A}}$ across our grids; its lower HR inflates the service term. Changing (β, δ, τ) mainly shifts points along a short diagonal; larger τ tends to reduce HR at similar quality (slower decisions). On this workload, local recency outperforms counter-based savings at the tested horizons.
- **LFU** improves with capacity (e.g., HR $\sim$0.77 at cap 2000, θ=0.65) but remains behind LRU/qLRU at matched quality and θ, indicating frequency is less predictive than recency for these prompt streams.

Summary and practical guidance.

1. *Recency is a strong prior.* LRU dominates HR and is frequently on the (Quality, $C_{\mathcal{A}}$) frontier, especially at small/medium capacity and higher θ.
2. *qLRU-ΔC (q=0.4) is the best cost knob.* At moderate thresholds and larger caches it achieves the lowest $C_{\mathcal{A}}$ with only a modest HR gap to LRU; use it when memory is abundant and theoretical cost is the objective.
3. *Duel and LFU are dominated on this trace* under the tested grids: Duel's lower HR keeps cost high; LFU's frequency signal adapts too slowly to non-stationarity.

4.3 Reproducibility

Setup. We use a Parquet file that already contains CLIP embeddings (loaded directly by `PromptDatasetManager`). All runs fix the unit regeneration cost to C_r=1 and use the θ-centered map 4.

Benchmark.

```
# 1) Run the benchmark
python benchmark_cache_policies.py \
  --data /path/to/diffusiondb.parquet \
  --policies LRU LFU QLRUDeltaC Duel \
  --capacities 20 200 1000 2000 \
  --thresholds 0.65 0.75 0.85 \
  --q-values 0.1 0.2 0.4 \
  --duel-beta 0.6 0.75 \
  --duel-delta 0.02 0.05 \
  --duel-tau 100 200
```

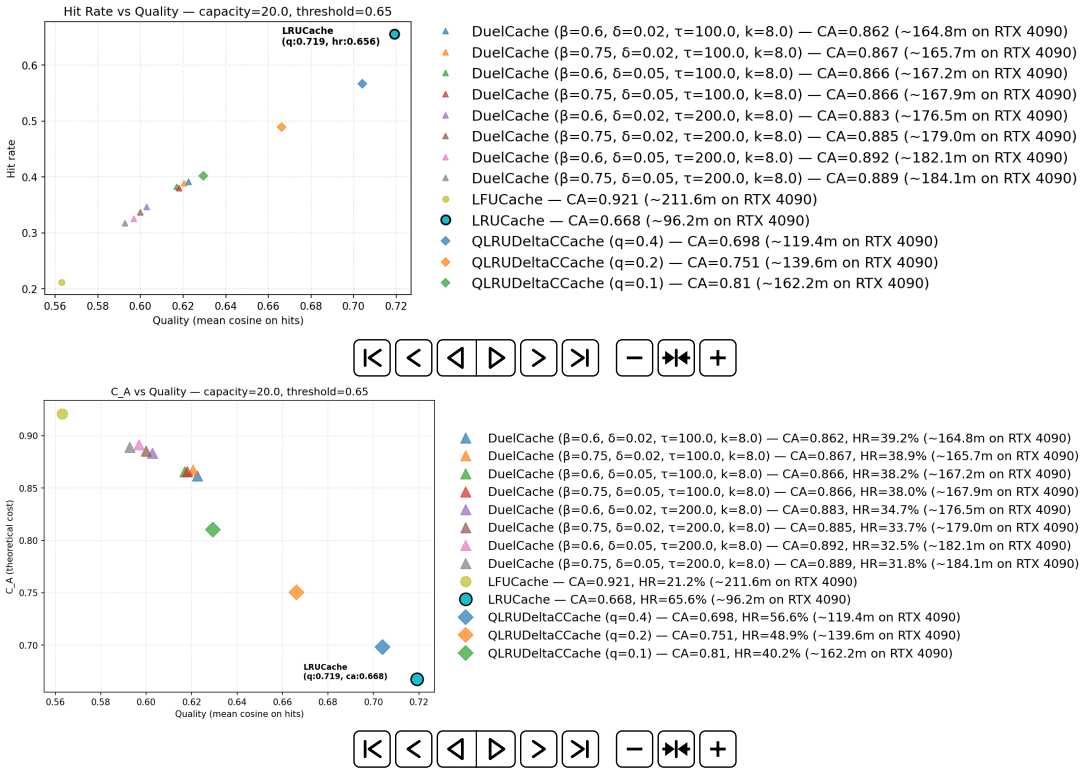


Figure 6: On top Hit Rate vs Quality, below $C_{\mathcal{A}}$ vs Quality experimental result. If animator not working well please refer to GitHub repo

Plots.

```
# 2) HR vs Quality
python plots_hr_quality.py --bench_dir Results/<STAMP>

# 3) C_A vs Quality
python plots_ca_quality.py --bench_dir Results/<STAMP>
```

These commands produce a run folder `Results/<STAMP>` with `summary.csv`, per-request `histories/`, and PNGs under `Results/<STAMP>/charts*` for HR–Quality and $C_{\mathcal{A}}$–Quality. We set RNG seeds (`-seed`) for reproducibility and pin library versions in the project's `requirements.txt`.

5 Conclusions

On DiffusionDB with CLIP embeddings, *recency* is a very strong prior. Across capacities (20–2000) and thresholds ($\theta \in \{0.65, 0.75, 0.85\}$):

- **LRU** consistently achieves the highest hit rate at a given quality and is on the (Quality, $C_{\mathcal{A}}$) frontier in most panels—especially at small/medium capacity or at higher θ.
- **qLRU-ΔC** (q=0.4) provides the best *theoretical cost* at moderate thresholds and large caches (e.g., at θ=0.65 and cap = 1000/2000 it outperforms LRU in $C_{\mathcal{A}}$ with only a modest HR gap), offering a clean knob to balance admission aggressiveness.
- **Duel** is dominated under our grids: lower HR inflates the service term, and the counter horizon (τ) does not translate into enough savings on this trace.
- **LFU** improves with capacity but remains behind LRU/qLRU-ΔC at matched θ, suggesting frequency adapts too slowly to the non-stationarity of prompts.

In short: use **LRU** as a strong default; switch to **qLRU-ΔC** (q=0.4) when minimizing $C_{\mathcal{A}}$ is the priority and memory is abundant.

Future Work. The present study establishes a first step toward making text-to-image diffusion serving more efficient through similarity caching, but many extensions are possible. One promising direction is the design of adaptive thresholds that adjust dynamically to workload intensity and user tolerance, rather than relying on a fixed similarity cut-off. Another is the investigation of cache replacement policies that exploit richer signals, such as prompt popularity, temporal trends, or reinforcement learning, to decide which results to retain. On the systems side, scaling the framework to distributed or tiered caches (e.g., GPU memory, disk, and CDN layers) would enable deployment in real-world serving infrastructures. Beyond images, the same methodology could be applied to more demanding modalities like text-to-video generation, or even to audio and multimodal outputs, where caching could provide even greater savings. Exploring these avenues would contribute to bridging the gap between theoretical cache models and practical large-scale generative services.

References

[1] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, "Zero-shot text-to-image generation," *Proceedings of ICML*, 2021.

[2] A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, and M. Chen, "Hierarchical text-conditional image generation with clip latents," 2022.

[3] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, "High-resolution image synthesis with latent diffusion models," *Proceedings of CVPR*, 2022.

[4] "Midjourney." Accessed: 2025-08-20.

[5] G. Neglia, M. Garetto, and E. Leonardi, "Similarity caching: Theory and algorithms," vol. 30, no. 2, pp. 475–486. Conference Name: IEEE/ACM Transactions on Networking.

[6] J. Ho, A. Jain, and P. Abbeel, "Denoising diffusion probabilistic models."

[7] A. Nichol and P. Dhariwal, "Improved denoising diffusion probabilistic models."

[8] C. Zhang, C. Zhang, M. Zhang, I. S. Kweon, and J. Kim, "Text-to-image diffusion models in generative AI: A survey."

[9] J. Ho and T. Salimans, "Classifier-free diffusion guidance."

[10] Pinecone Systems, Inc., "Gptcache: Semantic cache for llms," 2023. Accessed: 2025-08-20.

[11] S. Agarwal, S. Mitra, S. Chakraborty, S. Karanam, K. Mukherjee, and S. K. Saini, "Approximate caching for efficiently serving text-to-image diffusion models," pp. 1173–1189.

[12] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever, "Learning transferable visual models from natural language supervision," in *Proceedings of ICML*, 2021.

[13] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *IEEE Transactions on Big Data*, 2019.

[14] Z. Wang, X. Chen, C.-H. Chan, H. Li, K. Ma, E. Xie, Y. Li, and P. Polo, "Diffusiondb: A large-scale prompt gallery dataset for text-to-image generative models," *arXiv preprint arXiv:2210.14896*, 2022.

[15] "Diffusiondb dataset on huggingface." `https://huggingface.co/datasets/poloclub/diffusiondb`. Accessed: 2025-08-24.

[16] "Diffusiondb project page." `https://poloclub.github.io/diffusiondb/`. Accessed: 2025-08-24.

[17] J. Walton, "Stable diffusion benchmarks: 45 nvidia, amd, and intel gpus compared," Dec. 2023. Tom's Hardware; RTX 4090 ~75 images/min at 512×512 (SD 1.5, TensorRT). Accessed 2025-08-24.